conference

proceedings

BSDCon 2002

San Francisco, California
February 11–14, 2002

Sponsored by

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

BSDCon 2002 San Francisco, California February 2002

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
USENIX URL: http://www.usenix.org
ALS URL: http://www.linuxshowcase.org

The price is $25 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$10 per copy for postage (via air printed matter).

ISBN 1-880446-02-2

Printed in the United States of America on 50% recycled paper, 10–15% post consumer waste.

USENIX Association

Proceedings of BSDCon 2002

February 11–14, 2002
San Francisco, California, USA

Conference Organizers

Program Chair

Sam Leffler, *Errno Consulting*

Program Committee

Chris G. Demetriou, *Broadcom Corp.*
Jun-ichiro itojun Hagino, *IIJ Research Laboratory/KAME Project*
Jordan K. Hubbard, *Apple Computer, Inc.*
Rob Kolstad, *Delos*
Perry E. Metzger, *Wasabi Systems, Inc.*
Jim Mock, *Consultant*
Ernest N. Prabhakar, *Apple Computer, Inc.*
Gregory Neil Shapiro, *Sendmail, Inc.*

Invited Talks Coordinator

Donn M. Seeley, *Wind River Systems, Inc.*

The USENIX Association Staff

External Reviewer

Marshall Kirk McKusick

BSDCon 2002

February 11–14, 2002
San Francisco, California, USA

Networking
Session Chair: Jun-Ichiro itojun Hagino, IIJ Research Laboratory/KAME Project

System Administration
Session Chair: Jordan K. Hubbard, Apple Computer, Inc.

Index of Authors

Message from the Conference Chair

Welcome to BSDCon 2002, held this year in San Francisco. This is the third BSDCon conference, but the first to be sponsored by the USENIX Association. The decision to organize the meeting in a refereed format for the first time, combined with the unexpected events of September 11, made for some unique challenges. Ultimately, however, thanks to the efforts of many people, we've ended up with a fine set of proceedings and an exciting conference program.

The refereed papers track is the result of the hard work of the program committee and, of course, all the authors who submitted their work for review. Each of the 30 submissions were read by at least three qualified reviewers, and every author received written comments. In the end, we accepted 15 papers, which are high quality and reflect the interesting work going on in the BSD community. Particular thanks are due to Chris Demetriou, Donn Seeley, and Greg Shapiro for their work in this process.

Donn Seeley did a tremendous job as Invited Talks Coordinator and provided written comments on all the submissions even though that was not his job! Dan Klein did his usual great work in organizing the tutorials. Thanks to Greg Shapiro, the program committee was able to meet at Sendmail, Inc., in Emeryville, California.

This meeting would not have happened without the dedication of the USENIX staff and the support of the USENIX Board of Directors. Thanks are especially due to USENIX Board Director Kirk McKusick, who championed picking BSDCon up as a USENIX-sponsored activity, and to Ellie Young, Matthew Lasar, and Jane-Ellen Long of the USENIX staff.

Sam Leffler
Program Chair

Porting NetBSD to the AMD x86-64* a case study in OS portability

Frank van der Linden
Wasabi Systems, Inc.
fvdl@wasabisystems.com

Abstract

NetBSD is known as a very portable operating system, currently running on 44 different architectures (12 different types of CPU). This paper takes a look at what has been done to make it portable, and how this has decreased the amount of effort needed to port NetBSD to a new architecture. The new AMD x86-64 architecture, of which the specifications were published at the end of 2000, with hardware to follow in 2002, is used as an example.

1 Portability

Supporting multiple platforms was a primary goal of the NetBSD project from the start. As NetBSD was ported to more and more platforms, the NetBSD kernel code was adapted to become more portable along the way.

1.1 General

Generally, code is shared between ports as much as possible. In NetBSD, it should always be considered if the code can be assumed to be useful on other architectures, present or future. If so, it is machine-independent and put it in an appropriate place in the source tree. When writing code that is intended to be machine-independent, and it contains conditional preprocessor statements depending on the architecture, then the code is likely wrong, or an extra abstraction layer is needed to get rid of these statements.

1.2 Types

Assumptions about the size of any type are not made. Assumptions made about type sizes on 32-bit platforms were a large problem when 64-bit platforms came around. Most of the problems of this kind had to be dealt with when NetBSD was ported to the DEC Alpha in 1994. A variation on this problem had to be dealt with with the UltraSPARC (sparc64) port in 1998, which is 64-bit, but big endian (vs. the little-endianness of the Alpha). When interacting with datastructures of a fixed size, such as on-disk metadata for filesystems, or datastructures directly interpreted by device hardware, explicitly sized types are used, such as *uint32_t, int8_t,* etc.

1.3 Device drivers

BSD originally was written with one target platform (PDP11, later VAX) in mind. Later, code for other platforms was added, and 4.4BSD contained code for 4 platforms. NetBSD is based on 4.4BSD, but has steadily expanded the number of supported platforms over the years. As more platforms were added, it became obvious that many used the same devices, only using different low-level methods to access the device registers and to handle DMA. This led to, for example, 5 different ports having 5 seperate drivers for a serial chip, containing nearly identical code. Obviously, this was not an acceptable situation, with ports to new hardware being added every few months.

To remedy this situation, the *bus_dma* and *bus_space* layers were created [1], [5]. The *bus_space* layer takes care of accessing device I/O space, and the *bus_dma* layer deals with DMA access. For each NetBSD port to a new architecture, these interfaces must be implemented for each I/O bus that the machine uses. Once that is done, all device drivers that attach to such an I/O bus should compile and work without any extra effort.

2 Machine-dependent parts

Of course, not all code can be shared. Some parts deal with platform-specific hardware, or simply need to use machine instructions that a compiler will never generate. A few userspace tools will also be platform spe-

*x86-64 is a trademark of Advanced Micro Devices, Inc.

cific. Here is a summary of he most important machine-dependent parts of NetBSD that need to be dealt with when porting NetBSD to a new platform.

2.1 Toolchain

First and foremost, a working cross-toolchain (compiler, assembler, linker, etc) is needed to bootstrap an operating system on to a new platform. The GNU toolchain has become the de-facto standard for open source systems, and NetBSD is no exception to that rule. Since the ELF binary format is used by almost all NetBSD ports (and should be used by any new ports), as well other operating systems such as Linux, making the GNU toolchain work for NetBSD usually involves not more work than creating/modifying a few configuration files. The exception being the case where the target CPU isn't supported at all yet, which makes for a much greater effort.

2.2 Boot code

The boot code deals with loading the kernel image in to RAM and executing it. It interacts with the firmware to load the image. The effort needed to write the boot code largely depends on the functionality offered by the firmware. Often, the limited capabilities of the firmware are only used to load a second stage bootloader, containing code that is more sophisticated in dealing with filesystems on which the kernel image may reside, and the file format(s) that the kernel may have.

2.3 Traps and interrupts

Trap and interrupt handling is obviously a highly machine-dependent part of the kernel. At the very least, the entry points for these must have machine-dependent code to save and restore CPU registers. Furthermore, CPUs will have different sets of traps, needing specific care.

2.4 Low-level VM / MMU handling

Memory management units (MMUs) tend to be quite different from CPU to CPU. They may even be different within one family of processors (for example, the PowerPC 4xx series has an MMU that differs significantly from the 6xx series). They may also be very different in what they expose to the hardware. MMUs may for example have a fixed page table structure that they walk in hardware, or leave many more operations up to the software, exposing translation lookaside buffer (TLB) misses. The low-level virtual memory code in all 4.4BSD-derived systems is called the *pmap* module, a name taken from the Mach operating system, whose virtual memory (VM) system was used in 4.4BSD.

2.5 Port-specific devices

Some platforms will have devices that are highly unlikely to appear on any other platform. These may include on-chip serial devices and clocks. Porting NetBSD to a new platform often involves writing a driver for at least one such device.

2.6 Bus-layer backend code

As mentioned above, the device code uses a machine-independent interface for I/O and DMA operations. The differences per platform are hidden underneath this interface, and the implementation of this interface deals with the platform specifics. This interface is heavily used in device drivers, so a small memory footprint and speed are important. Often, the interface is implemented as a set of macros or inline functions.

2.7 Libraries

In userspace, the C startup code and a few libraries will contain machine-dependent code. The library parts in question are mainly the system call interface in the C library, the optimized string functions in the same library, and specific floating point handling in the math library. There are a few other, lesser used areas, like the KVM library which deals with reading kernel memory. Lastly, shared library handling (relocation types) will likely be different from other platforms.

3 The x86-64 hardware

Before going into the specifics, a brief introduction to the AMD x86-64 architecture[2]is in order. The AMD x86-64 (codenamed "Hammer") architecture specification was released end of 2000. As of December 2001, no hardware implentation was publicly available yet (NetBSD/x86-64 was exclusively developed on the Simics x86-64 simulator made by VirtuTech). Since x86-64 is essentially an extension to the IA32 (or *i386*, as it is known in NetBSD) architecture, its predecessor will be introduced first below.

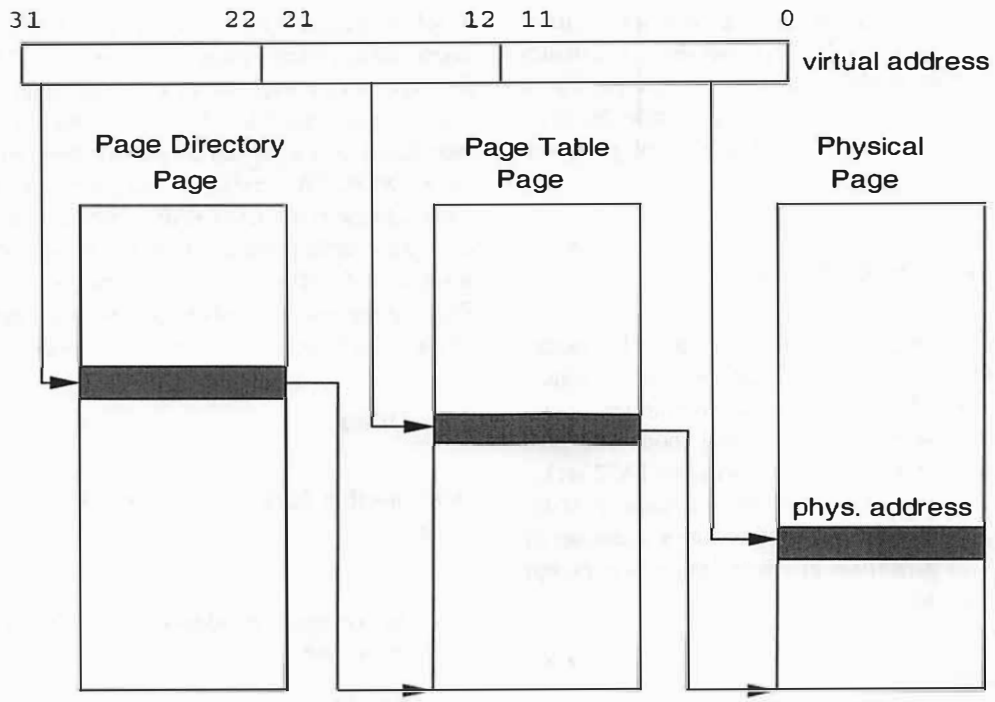

Figure 1: IA32 virtual address translation (4K pagesize). The entries in the page table and page directory are 32 bits wide.

3.1 The IA32 architecture

IA32 is the name that Intel gave to the 32-bit architecture that was originally introduced with its 80386 CPU, and has, through its application in the PC world, become the most popular CPU architecture today.

IA32 CPUs have the following features [4]:

- Seven 32-bit wide general purpose registers, 4 of which can also be used in 16-bit and 8-bit chunks
- A large set of instructions.
- From the 80486 and up, an on-chip floating point unit.
- From the Pentium III and up, Streaming SIMD Extensions (SSE), a class of instructions dealing with parallelized load/store and computation, targeted at graphics applications. SSE also adds a set of new registers, 64 bits wide, for use with SSE instructions. The Pentium 4 added yet more SSE instructions, and widended the SSE registers to 128 bits. These newer SSE instructions are known as SSE2.
- 32-bit wide addressing
- An MMU supporting the usual page protection schemes, and four gigabytes of virtual memory through a three-level pagetable. Different parts of a virtual address are used as indices into page table structures. These structures contain some information on the protection of the page, and point to the physical address of the page in which the structure to be indexed at the next (lower) level is contained. The lowest level of page tables contains the actual physical address to be referenced. Later CPUs also support an extended version of this scheme called Physical Address Extensions (PAE), enabling the use of more than 4G of physical memory. Virtual address translation for the IA32 architecture is shown in figure 1.
- Memory segmentation through descriptors, describing the type, base address and length of a section of memory. Memory descriptors (and other types) are stored in special tables.
- Segment registers that specify which descriptor is used to determine the location of loads/stores/execution.
- Trap/interrupt handling through a special array of descriptors, called the Interrupt Descriptor Table (IDT).
- Four different execution modes for programs: plain 16-bit mode ("real mode", backward compatible to the 8086); 16-bit "protected mode" (16-bit mode

with protection); 32-bit "protected mode" (this is the mode that all modern systems use); and finally "virtual 8086 mode", which runs old-style 16-bit programs in a virtual "real mode", while the operating system is actually running in 32-bit protected mode.

3.2 General x86-64 extensions

The x86-64 architecture is essentially a 64-bit extension of the IA32 architecture. In addition to the legacy "real" (16-bit) and "protected" (32-bit) modes, it defines a "long" (64-bit) mode. In real mode and protected mode, it is fully compatible with the IA32 architecture. In long mode, it is capable of running 32-bit binaries without modification, and contains a number of extensions. This paper only discusses long mode, except where explicitly noted otherwise.

3.3 Registers

The general-purpose registers that already were present in the IA32 architecture were extended to 64 bits. Eight general purpose registers were added, yielding a total of fifteen (not counting the *esp* register, see figure 2). This addresses an often heard complaint about the IA32 architecture: it has very few general purpose registers. Additionally, eight SSE2 registers were added. For consistency, all general purpose registers can have their lower 16 bits or lower 8 bits addressed specifically in instructions, something that was only possible for four of the registers in the IA32 architecture.

To be backward compatible, computations and moves involving the lower 16- and 8-bit parts of the registers do not affect the upper bits. However, 32-bit operations are zero-extended. A special 64-bit immediate register move instruction was added to conveniently use 64-bit constants; 64-bit immediate values are not allowed in other instructions.

3.4 Memory management

The idea of the x86-64 being an extended IA32 architecture is also reflected in its memory management unit (MMU). The x86-64 has a 64-bit virtual address space, however, initial implementations of the architecture are specified to only use 48 out of these 64 bits, and 40 bits of physical address space. Addresses are specified to be sign-extended, essentially leading to a "hole" in virtual memory space that cannot be addressed. To enable the MMU to handle the translation of 48 bits of virtual address space, an extra page table level was added, in addition to the extra level that already had been added in later implementations of the IA32 architecture to support PAE (see figure 3, the dotted line inside the virtual address shows where PAE ends). So basically, the x86-64 page table scheme is the IA32 PAE scheme, with 512 instead of 4 page table pointer directory entries, plus a fourth level, called PML4. This similarity goes so far that the PAE feature must actually be specifically enabled as part of the steps to get the chip into long mode.

3.5 Other

Other notable features of the x86-64 architecture include:

- The possibility of addressing relative to the instruction pointer.
- Flat address space (the memory offsets specified through the code and data segment registers are ignored).
- Most hardware (system) data structures were extended to hold 64-bit addresses where needed.
- Fast, special cased instructions for system calls into the operating system. AMD had already introduced these with the K6 CPU, and they were extended to 64 bits.
- A few special registers were added, such as the registers that hold the entry point for the SYSCALL/SYSRET instructions, and the EFER, the Extended Feature Enable Register, which, amongst other things, contains a bit that enables long mode.

4 The actual port

Using the list of machine-dependent operating system parts in section 2, the work that was needed to port NetBSD to the x86-64 architecture will now be discussed.

4.1 Toolchain

When the work on NetBSD/x86_64 was started, the GNU toolchain already had some basic x86-64 support in it, which had been developed for Linux at SuSe, Inc. Shared library support wasn't working yet, but

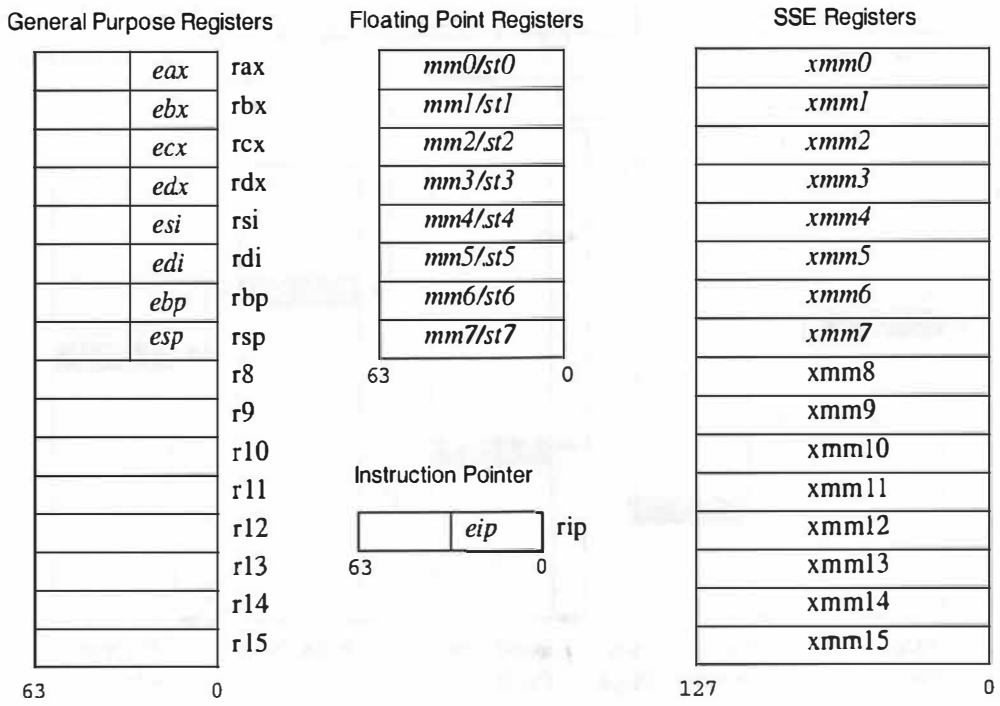

Figure 2: x86-64 registers. IA32 compatible registers are shown in italic

compiling, assembling and linking applications mostly worked. The application binary interface (ABI) had also been defined[3]. Adapting this code for NetBSD came down to just modifying/creating some configuration header files. Naturally, a few compiler and linker bugs were present in the OS-independent code of the toolchain, but this could be expected as the x86-64 code was quite young.

There are some ABI issues to consider. The x86-64 ABI defines four non-PIC code models:

- Small. All symbols in the program are assumed to be at virtual addresses in 32-bit range.
- Kernel. As above, but instead, all addresses are expected to be in negative 32-bit range, i.e. in the upper 32 bits of 64-bit memory. Kernels are often run in the upper region of virtual memory, and this code model was added to map a kernel in that area, without having to specify full 64-bit offsets in the code.
- Medium. Size and address of the code segment are expected to be in 32-bit range, but data can be the full 64-bit range
- Large. No restrictions on data or code addresses. The compiler has to generate code to only use indirect addressing via registers to be sure that the full 64-bit range is addressable.

There are similar models for position-independent code. By default, userspace programs are expected to use the "small" code model. Other code models weren't yet completely supported when the port was done, although at least the "large" model turned out to be stable after a few small modifications, and was used for the kernel.

4.2 Bootcode and bootstrap

Since there is no actual x86-64 hardware available yet, and no definitive firmware interface for the upcoming x86-64 machines has been specified yet, the bootcode had to deal with whatever the simulator provided. The simulator provided a normal PC BIOS interface, meaning that the NetBSD/i386 bootcode could almost be used as-is. However, the kernel image to be loaded is a 64-bit ELF binary for the x86-64 case. Making this work was a trivial modification, since the code to load 64-bit ELF binaries had already been made machine-independent and placed into the stand-alone library used by the bootcode of various NetBSD platforms.

The initial bootstrap code in the kernel (i.e. the first code to be executed in the kernel) naturally had

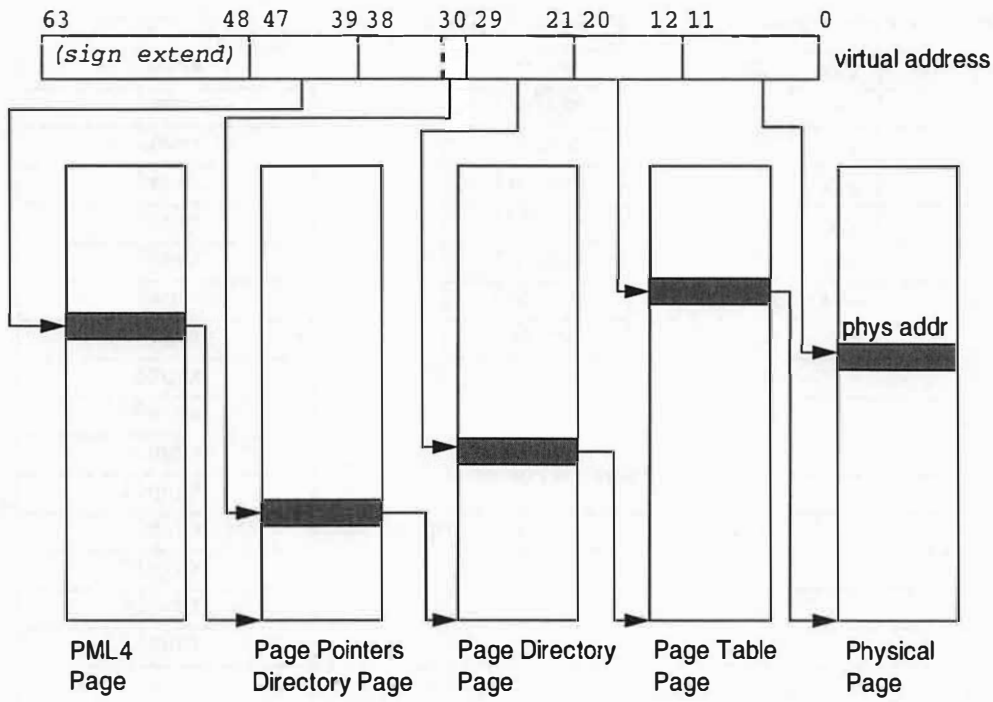

Figure 3: Virtual address translation in the x86-64 (4K pagesize). The entries in the page table structures are 64 bits wide.

to be written from scratch for this new NetBSD port, although it can be viewed as an extended version of that in NetBSD/i386. Since the x86-64 is fully IA32 compatible at power-on, it also needs to deal with getting the CPU out of 16-bit mode, in to 32-bit mode, and this time a couple of steps further, into 64-bit mode. The actual steps are:

1. Enable Physical Address Extensions
2. Set the LME (Long Mode Enable) bit in the EFER register
3. Point the %cr3 register at a prefabricated initial 4-level page table structure
4. Enable paging
5. The CPU is now running in a 32-bit compatibility segment. Fabricate a temporary Global Descriptor Table with a Long Mode memory segment, and jump to that segment
6. Because the kernel is mapped in the upper regions of memory, we could previously not address that region, as it lies well out of 32-bit range. But since we are now finally running in long mode, it is within range of the jump instructions, so use one to finally start executing the actual kernel code.

4.3 Traps and interrupts

The structure of the low-level trap and interrupt code is similar to that of NetBSD/i386, although no code could be shared. The x86-64 also uses an Interrupt Descriptor Table (IDT) to set up the vectors for traps and interrupts. The work done at the entry points for the traps was the normal save registers/dispatch/restore registers. Although the higher-level trap code can probably be shared between NetBSD/i386 and NetBSD/x86-64, since the set of traps is the same between the architectures, this has not yet been done.

Another set of traps is the system call entry points. The x86-64 supports the same mechanisms that were already present in the IA32 architecture: entering the kernel via a software interrupt, or doing so by issuing a call to a special type of structure (a *call gate*), which automatically switches to the kernel environment. These instructions perform some actions that are usually not needed in an environment where the address space is flat (i.e. the full 4G of virtual memory is available to programs in one chunk, with the kernel usually occupying the upper region). The SYSCALL and SYSRET instructions are optimized for this case, and can be used to implement a faster system call path, which can be an important factor in the performance of applications. The code to handle this was written, but not yet integrated;

currently the old-style entry points are still used, but this will change in the near future.

4.4 Low-level VM / MMU handling

The x86-64 MMU uses a page table structure that is very similar to the IA32. It basically is the IA32 with a Physical Address Extensions page table, extrapolated to have 4 levels, to deal with the 48 bits of virtual memory that the initial family of x86-64 processors will have. Because of this similarity, the i386 pmap module was taken, and abstracted to implement a generic N-level IA32 page table, with either 32- or 64-bit wide entries. The resulting code has been tested on both the x86-64 and i386 ports of NetBSD, with success. The differences between the IA32 and x86-64 code were hidden in C preprocessor macros and type definitions. The advantage of this approach is that it is now easy to optionally support PAE on the NetBSD/i386 as well, because this only means conditionally compiling in a few other macros and type definitions.

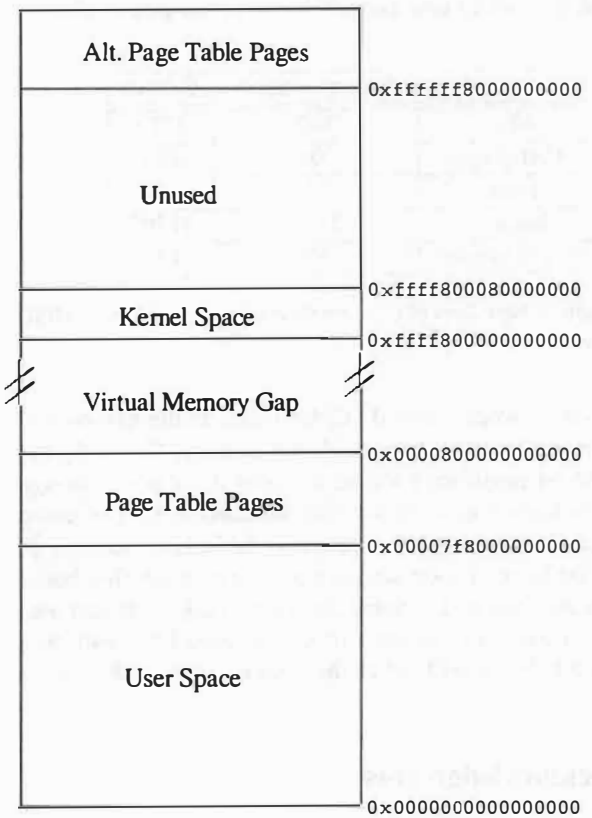

Figure 4: NetBSD/x86-64 virtual memory layout.

The implemented virtual memory layout is shown in figure 4. Because of the sign-extension that the CPU performs on virtual addresses, an unaddressable gap exists between $2^{47}$and $2^{64} - 2^{47}$. This is not unusual; such a gap is also found on e.g. SPARCv9 and Alpha processors. The memory layout is more or less a stretched-out version of the IA32 memory map, as was to be expected in a merged pmap module. A user process runs in the bottom half of virtual memory, while the kernel is always mapped in the top half. Part of the top half is unused, because the kernel doesn't need the huge amount of virtual memory available there, and it turned out that using such an amount of space blew up some data structures in a disproportional way. The upper part of the bottom half of virtual memory is taken up by recursively mapped page table pages, as is the the upper part of the top half of virtual memory (used if the page tables of a process other than the current one need to be changed).

This layout meant that the kernel was out of range of the "kernel" code model, specified in the ABI. The "large" model was needed, but not yet supported by gcc. Fortunately, it turned out that it did work, with a few small modifications. The NetBSD/x86_64 kernel will likely be changed to use the kernel ABI model, once real hardware is available, and a speed assessment will be made. This layout is being used for now because it is a consistent extension of the IA32 model, making pmap code sharing easier.

4.5 Bus-layer backend code

Much of the bus-layer backend code could be reused from the i386 port. For the PIO case, the instructions remained the same, so no changes were needed, except for some modifications to make it fit the extended 64-bit register set. The same goes for memory-mapped I/O. The DMA framework needed to deal with possibly having 32-bit PCI, which would be unable to do DMA access into memory above the 4G limit. For now, a simple solution was picked of having DMA memory for 32-bit PCI always come from below the 4G limit. This needs to be revisited later; for machines with more than 4G of memory conditions may occur where RAM is available, but not below the 4G limit. To avoid this problem, *bounce buffers* can be introduced; they are temporary buffers on which the DMA is done, and to or from which the data must be copied to its actual location.

4.6 Port-specific devices

So far, the NetBSD x86-64 port does not deal with any platform-specific devices. The simulator simulates a

number of hardware components that is known from the PC world (like the host-PCI bridge, etc). These components "just worked", and no modifications were needed, after the *bus_space* and *bus_dma* layers were implemented.

4.7 Libraries

The main work in userspace was porting the libraries and C startup code. The C startup code and C library were fairly trivial to port. Most of the work that had to be done was to write the system call stubs and the optimized string functions, keeping in mind that the x86-64 ABI passes most arguments in registers, instead of always on the stack as the i386 ABI specifies. The math library was a bit more work. It could share a lot of code with the i386 (really i387) code, since the FPU has the same instructions, but there was an ABI difference. The x86-64 ABI specifies that floating point arguments are passed in SSE registers, but the i386 ABI passes these on the stack. A few macros had to be written to extract the arguments to the various (mostly trigonometry) functions, prepare them, and then use a common bit of code for both the i386 and x86-64 ports. Lastly, the dynamic linker had to be adapted to deal with the types of relocation that x86-64 shared libraries may use.

4.8 Compatibility code

The x86-64 offers the option to run 32-bit i386 applications without modification. This is a useful option, as it enabled operating systems to run older applications out of the box. Some support for this is needed in the kernel, though. The basic item that is needed to run a 32-bit application is to install 32-bit compatibility memory segments in the various descriptor tables of the CPU. The CPU will execute instructions from such a segment in a 32-bit environment. However, traps to the kernel will switch the CPU to 64-bit mode. This has the advantage that there doesn't need to be any special kernel entry/exit code for 32-bit applications. 32-bit programs do have a different interface to the kernel; they pass arguments to system calls in different ways, and in 32-bit quantities. Also, structures passed to the kernels (or rather, the pointers to them) will have a different alignment. These issues needed to be addressed to enable running old NetBSD/i386 binaries.

Compatibility code to run binaries from various platforms (Linux, Tru64, Solaris, etc.) has been a part of NetBSD for a long time. So, not surprisingly, this issue had already been tackled once before, when NetBSD/sparc64 needed to deal with running 32-bit SPARC binaries. This code, the *compat_netbsd32* module, implements a small layer which translates (if needed) 32-bit arguments to system calls to their 64-bit counterparts.

5 Conclusions and future work

The port of NetBSD to AMD's x86-64 architecture was done in six weeks, which confirms NetBSD's reputation as being a very portable operating system. One week was spent setting up the cross-toolchain and reading the x86-64 specifications, three weeks were spent writing the kernel code, one week was spent writing the userspace code, and one week testing and debugging it all. No problems were observed in any of the machine-independent parts of the kernel during test runs; all (simulated) device drivers, file systems, etc, worked without modification.

The porting effort went smoothly. Table 1 shows the amount of new code written. In the area of sharing

Area	Assembly Lines	C lines
libc	310	2772
C startup	0	104
libm	52	0
kernel	3314	17392
dynamic linker	59	172

Table 1: New lines of C/assembly code per area of the NetBSD source tree

code between "related" CPUs (such as the x86-64 and the IA32), some more work can be done. Currently, the x86-64 pmap isn't shared between the 2 ports, though it is known to work for both architectures. The pmap code is counted as "new code" in table 1, but most of its 3500 lines of code was in some form or another based on the i386 code. Some descriptor table code can also be shared. The number of new C code lines will drop well below 10,000 when the code is properly shared.

Acknowledgments

This work was paid for by my employer, Wasabi Systems, Inc. I'd also like to thank AMD for their support, Virtutech for the simulator, and the folks at SuSe for doing the Linux toolchain work.

References

[1] Jason Thorpe: A Machine-Independent DMA Framework for NetBSD, Usenix 1998 Annual technical conference.

[2] Advanced Micro Devices, Inc: The AMD x86-64 Architecture Programmers Overview, http://www.amd.com/products/cpg/64bit/pdf/x86-64_overview.pdf

[3] Hubicka, Jaeger, Mitchell: x86-64 draft ABI, http://x86-64.org/abi.pdf

[4] Intel Corporation: Pentium 4 manuals, http://developer.intel.com/design/Pentium4/manuals/

[5] Chris Demetriou: NetBSD bus_space(9) manual page, originally in NetBSD 1.3, 1997.

Problems updating FreeBSD's card system from ISA to PCI

M. Warner Losh
Timing Solutions, Inc
Boulder, Co
imp@village.org

Abstract

FreeBSD's 16-bit PC Card implementation has used the ISA legacy interface. PCI support was added by making PCI-CardBus and PCI-PCMCIA bridges behave in ISA compatibility mode. While this technique worked in laptops, it made support of add-in PCI cards with CardBus or PCMCIA bridges impossible. PCI-PC Card bridges are unlike traditional devices because they can have connections to multiple busses, offering both ISA and PCI interrupt routing for them and any devices connected to them. Expanding support to add-in PCI cards with 16-bit PC Cards connected exposed weaknesses in the PC Card implementation of FreeBSD as well as other parts of the system. Vendor BIOS quality, variance in hardware implementation details from standard and weaknesses in the FreeBSD development model made incorporation of these improvements into FreeBSD 4.4-RELEASE difficult. Lessons learned will be incorporated into the 32-bit CardBus support forthcoming in FreeBSD 5.0.

1 Problem

FreeBSD's[FreeBSD] PC Card implementation in FreeBSD 4.3-RELEASE and earlier only routed ISA interrupts to 16-bit PC Card cards. It configured PCI-PC Card bridges to look like old ISA devices, complete with ISA interrupt routing. This strategy worked well for Laptops where the ISA interrupt signals were available to the PCI bridge. However, for add-in PCI-PC Card bridge cards, this strategy failed because the add-in cards do not have connections to the ISA bus' interrupts. Since there was no connection to the ISA interrupts, these add-in cards could not be configured to use them. The growing popularity of these cards in new PCI only systems was a problem. Additionally, ISA interrupts cannot be shared in the absence of specialized hardware which laptops lack, making some laptops very difficult to configure. Since ISA interrupts are also hard to detect for devices that do not have drivers configured for them, mysterious failures happened frequently when a novice user would configure a FreeBSD's PC Card system improperly.

An effort was made to allow PCI routing of interrupts on PCI bridges, and for sharing of interrupts of 16-bit PC Cards when they are connected to a PCI-PC Card bridge. No attempt was made to expand FreeBSD support to include 32-bit CardBus cards as part of this effort. As a result of this effort, 16-bit PC Cards in add-in PCI-PC Card bridge cards now work, as do most of the bridges supported by the previous code. In addition, the support load from improperly configured laptops has dropped with the new automatic interrupt configuration.

2 Implementation History

Prior to the 4.4-Release, FreeBSD's PC Card implementation relied on treating all PC Card[PC Card] bridges as ISA devices. FreeBSD's implementation was written in a time before PCI[PCI] devices that supported PC Card were widely available. Two basic types of PCI bridges have appeared. One is a 16-bit PC Card bridge and called a PCI to PCMCIA bridge. The other is for 16 or 32-bit cards and called a PCI to CardBus bridge. While most of these bridges support a standard configuration space and register set, some older bridges do not and need varying amounts of special case code. This paper uses the term PCI-PC Card bridge to generically refer to any of these bridges.

FreeBSD's PC Card code was able to treat these PCI-PC Card bridges as ISA devices because they

generally were connected to the ISA bus or ISA bus' interrupts in the laptop configuration. Since laptop support was the primary target of FreeBSD's PC Card code, little attention was paid to the add-in card problem since most machines had an ISA slot, which was well supported. Microsoft's hardware design guides have ensured that laptops manufacturers made these connections to the ISA bus. FreeBSD was able to program these PCI devices in such a way as to ensure that the ISA compatibility worked for laptop users, at the price that the laptop user would have to hand configure each laptop to list those interrupts not used by other hardware.

In recent years, a number of trends in the industry has made ignoring the add-in card problem less and less appealing. The first trend was toward more and more desktop or server machines having PC Card in them at all. Many users desired to read flash cards from their digital cameras on desktops. One of the solutions to do this was to install PC card slots and insert the flash cards into the PC Card slots, possibly with an adapter. A second reason for having PC Card slots in a desktop or server machine was for wireless support. While some of the early, pre–802.11 wireless cards were available in PC Card, ISA and PCI versions, nearly all of the recent 802.11[WiFi] and 802.11b cards have only PC Card versions. To place an 802.11b wireless card into a desktop or server machine, you needed to have some type of PC Card bridge to provide the slots to insert the wireless card into. ISA, PCI and other solutions exist to solve these needs. The PCI add-in card forced FreeBSD wireless PC users to use ISA PC Card bridges. This solution was adequate for a long time, but other trends made it less viable.

The second trend was the retirement of the ISA bus expansion from PC compatible computers. The number of ISA slots in system had been shrinking for years as the superior PCI bus provided a number of advantages. Microsoft had a vested interest in retiring ISA support from Windows, so created the PC 98 System Design Guide (not to be confused with the PC-98[PC-98] machines made by NEC and sold in Japan). To be compliant with the PC 98 Design Guide, a system could not have any ISA expansion bus slots if it was manufactured after a certain date. Subsequent design guides[PC99, PC2001] reaffirmed this restriction. Many systems now omit ISA bus expansion slots. FreeBSD users that wanted to connect wireless cards to these system were left without a general solution (although specialized solutions remained available to PCI only systems).

The third trend has been towards integration of more and more devices into laptops which can cause a decreased availability of unused interrupts. In some generations of laptops, each of these devices would use an interrupt. Since FreeBSD couldn't share PC Card interrupts with other devices, some machines could not find enough interrupts to support multiple PC cards. Some laptops also could not reserve an interrupt for card change status events. As the number of free interrupts on the typical laptop has declined, the difficulty and complexity of correctly configuring the system grew. While FreeBSD has tried to prevent the user from reusing an interrupt, it cannot protect against devices where no driver has attached and there's no way of knowing which resources are consumed. Laptops release in the last year or two have shown a trend away from ISA devices in favor of PCI devices, which automatically configure resources and which can share IRQs. While the number of devices on laptops continues to grow, their impact has been lessened somewhat, leaving only the problems of highly integrated laptops from a few years ago.

3 Prior Implementation Details

The FreeBSD PC Card implementation prior to 4.4–RELEASE was done in two parts. One part of the implementation was done in the kernel, while another part was done in a user-land daemon.

When the system boots, the kernel detects PCI-PC Card bridges and programs them into ISA compatibility mode. Sometimes, for proper operation, special settings in the BIOS are required. A separate part of the kernel then detects the ISA devices (or the PCI ones in ISA compatibility mode). On attach, things are setup to detect cards arriving into the system, using the traditional ISA interface. This mode uses only the ExCA registers, defined in the PCIC standard, to manipulate the bridge.

When a card is inserted into a PC Card slot, the kernel does a basic power up of the card, and notifies pccardd that a card status has changed. Once the card is ready, pccardd will read its meta data, which it uses to configure the card. It then resets the card, and tells the kernel which resources the card requires, and what driver to attach to the card. The kernel then asks the driver if it really supports the card, and if so, adds it to the FreeBSD device

tree. The pccardd daemon is then informed of the success of the operation. Any final user-land configuration of the new device will follow. When a card is ejected, the kernel will tell detach the device and tell pccardd that the device is gone. A program exists to query the state of the PC Card system, read the card's meta data and to force configuration of a card exists.

During the configuration process, both the kernel and pccardd will manipulate the hardware in a number of ways. They set bits in the hardware to cause certain things to happen, then polls the bits for a response. So as to allow other system activity in the interim, there is a small sleep between attempts to check the bits. This polling works well in an ISA system. If an interrupt handler fails to clear the interrupt condition, nothing happens for the typical ISA device (except maybe to miss future interrupts) since the ISA bus uses edge triggered interrupts. While FreeBSD is manipulating the hardware to configure a PC Card, various interrupts may occur. The FreeBSD implementation didn't need to handle these interrupts to appear to be working, since they were one shots.

4 Important Hardware Details

A number of problems present themselves when this solution is attempted. However, to understand them one must understand how PCI hardware works, what 16-bit PC Cards do at each step of their configuration process, and how the PCI-PC Card bridges react. Some of these items are well documented, while others are observed behavior.

The PCI bus differs from the ISA bus in many ways. Unlike the ISA bus, the PCI bus has level sensitive interrupts. The PCI bus also allows for interrupt sharing. The PCI bus defines 4 signals for interrupts that are routed to each device on the PCI bus, **INTA#**, **INTB#**, **INTC#** and **INTD#**. The PCI device can pick one it will use (although that choice is usually hardwired to **INTA#**). When a PCI device asserts an interrupt, it stays asserted until the interrupting condition is cleared. Failure to clear an interrupt condition in the device's ISR will cause the PCI **INTx#** line to remain asserted. Since the the interrupt remains when the ISR exists, it will be called again. Lather. Rinse. Repeat. If the ISR never clears the condition, this causes an interrupt storm. This will cause the system to loop forever in a wedged, or semi-wedged state. If one fails to clear an interrupt condition on an ISA device, no such pathological behavior happens.

PCI-PC Card bridges assert two types of interrupts. The first type is for card status changes, while the second is for function interrupts. The first type can be masked, but the card function cannot be masked. When a PC Card indicates that an interrupt is present, the PCI interrupt line is asserted. Short of instructing the bridge to use ISA signaling for card function interrupts, there is no way to mask this interrupt. Every family of PCI-PC Card bridges have their own method to control how interrupts are routed, which adds complication. The implications of this are that both the bridge driver and children drivers of the bridge must be more careful in their interrupt handling. PCI-PC Card bridges do not offer a way to mask the card function interrupts. Driver writers must ensure that an ISR is present before the card begins to generate interrupts.

A 16-bit card can be reset by writing certain bits to a configuration register (sometimes called a COR reset). When the card is in this reset state, it's **READY** pin will sometimes be pulled low to indicate that the card is not ready (or that the card has finished resetting, depending on the card). The **READY** pin is shared with the **IREQ#** pin for 16-bit PC Cards. The **IREQ#** pin is active low and indicates that the card is interrupting. This signal is held low until the reset bits are turned back off. The COR register varies in location from PC Card to PC Card, and is contained in the CIS (or meta data) that is parsed by pccardd.

When powering a 16-bit card up, an interrupt is generated when the power sequence is complete on many (but apparently not all) PC Card bridges. Driver writers are expected to detect this interrupt, acknowledge it, and continue with their power on sequence.

5 Naive Implementation

A naive implementation strategy to add PCI interrupts to FreeBSD 16-bit implementation would be to create a real PCI attachment for the PC Card bridges, program the PCI bridge into "native"

mode, but otherwise leave the previous implementation alone. The author tried a naive implementation in full ignorance of the problems he faced in doing so.

The experienced reader may have already notice several pitfalls from the above descriptions. Nearly all of the problems result in system hangs due to interrupt storms. Some problems are less obvious and more failsafe, but can result in no PC Cards working.

Each of the above problems posed a hurdle in upgrading FreeBSD's PC Card implementation. These problems were dealt with in two ways. Many of the busy wait portions of the code were rewritten and moved entirely into the kernel. They were made interrupt driven and each of the interrupts were acknowledge. These were the easy ones to fix (once you discovered which magic register the interrupt source bits were stored in, as there are more than one). The hard part of fixing these bugs was always finding sufficient documentation to be sure that the interrupt bits in question were the right ones.

Two of the issues were hard to fix and merit further discussion. The first issue is that of the COR reset. In the user-land approach, all knowledge of where to write the magic bits rests in pccardd. Indeed, when level interrupts weren't an issue, it made good sense to have all the knowledge there. The normal PC card bus driver interfaces allowed the right part of the attribute memory to be mapped at the right times to write to the COR. pccardd didn't have to manage that at all, it just would seek to the right location; write to set the reset bit; wait and then seek/write to clear the reset bit. Moving all this functionality into the kernel proved to be difficult. The kernel would have to be told where the COR was located, and the ISR routine would need to write the right value to the COR. Since the old implementation made this tedious to do, I elected to fix the problem in another way. Examination of the BSD/OS, Linux[Linux-CS] and NetBSD[NetBSD] PC Card code showed that none of them had a COR reset at all. It appears to be unique to FreeBSD's PC Card implementation. I eliminated it and so far no ill effects have be traced to its elimination.

The second issue was PC Card function interrupts. During the configuration process, these were asserted at unpredictable times. Each card appears to have its own way of resetting these condition during startup. Since the PC Card bus driver had no knowledge of the plurality of card dependent methods, it could not acknowledge the interrupt, or otherwise force it to go away, thus causing an interrupt storm. The solution adopted by the author was to route the function interrupts to the ISA bus until the driver had attached an interrupt status routine. This solved the transient problem, but may be the cause of strange system hangs on reboot.

6 PCI Interrupt Routing

Once the simple problems were solved, I deployed my code on a number of laptops and desktop system with a variety of chip-sets. The results showed that addition hurdles awaited. However, to understand them, some basic hardware and system boot strap issues must be discussed.

The biggest problem encountered was PCI interrupt routing. On the laptops I had done the initial development the BIOS automatically routed an interrupt to the PCI - Cardbus bridge. It turns out that the BIOS is not required to route the interrupts, and the OS is responsible for doing this in some cases.

The PCI interrupt routing problem is thornier than it may appear on the surface. It is generally not the case that all interrupts can be directed to any pin on any card on the PCI bus. There are usually significant restrictions on which interrupts can be used. These restrictions come from two different sources. First, most PCI Cards are wired using the so-called barber pole arrangement, where **INTA#** of slot 1 is connected to **INTB#** of slot 2, **INTC#** of slot 3, **INTD#** of slot 4, **INTA#** of slot 5, etc. The standard doesn't specify the arrangement, so others exist. These pins are wire-ored together, so they can only be connected to one interrupt at the bus controller. If one is routing interrupts for a given device, and another device has routed an interrupt on the wire that, there is no choice in which interrupt gets routed. The two devices must share that interrupt. Since the wiring diagram of a PCI bus can be arbitrary and complex, the Operating System must have knowledge of this wiring diagram.

The second source of problems is controlling the bridges between the device and the CPU. They may have restrictions on which interrupts can be used, and each chip-set has a slightly different API to route the interrupts. Also, exact knowledge of how

the bridges are wired together is often required to program the bridges correctly. The number of different bridge chip-sets found in deployed computers is large. Far larger than one could hope to ever support, even if one could get the wiring knowledge necessary to do the routing, which isn't possible in the absence of auxiliary tables.

To bring order to this chaotic state of affairs, the PCI SIG standardized the BIOS API in the PCI BIOS Specification[PCI BIOS]. This standard includes way to access each PCI Device's config space. version 2.1 added the ability to query $PIR Interrupt Routing Table and to route interrupts. These functions made it possible to do the routing.

Mike Smith wrote a fairly complete implementation of PCI interrupt routing using PCI BIOS. However, his implementation was for the forth coming FreeBSD 5.0, and not for FreeBSD 4.x which my PC Card implementation was targeted at. After porting most of the PCI interrupt code to 4.x, I was able to use many of the machines that had previously had unassigned interrupts for the PC Card bridge. This new interrupt routing was not without its faults.

6.1 Calling the BIOS

The PCI BIOS Specification defines two different ways one could call the PCI BIOS. The OS can look for a special signature in the BIOS ROM area of the computer (Physical address 0E0000h - 0FFFFFFh). Once it finds the signature, the entry point address is located with the entry and can be used to call the PCI BIOS using a standard CALL FAR (callf) instruction. The OS uses the BIOS32 Service Directory to obtain information to build the proper segments for this entry point.

The PCI BIOS also provides a 16-bit real and protected mode interface. The int 1Ah software interrupt is used to access the 16-bit interface, and operates in either real or protected mode. These BIOS functions may also be accessed by calling the industry standard address for int 1Ah (physical address 0FFE6Eh) by simulating a INT instruction (pushf followed by callf).

Unfortunately, the PCI BIOS standard doesn't appear to mandate that both of these interfaces be available to the OS. The PC97 Design Guide appears to rectify this situation by stipulating that both interfaces must be supported. However, the PC 98 Design guide required all machines to migrate towards using ACPI to configure their interrupts, a trend that continued in subsequent Design guides. As such, extremely new machines have also started to exhibit problems with PCI BIOS configuring the interrupts.

Unfortunately, the standard doesn't appear to mandate that all calling methods actually work. With some BIOSes, only a subset of the calling methods are available. If you are lucky, the one you want will work. Some BIOSes appear to support the BIOS calls, but later will perform an illegal access inside the BIOS call. On machines newer than about 1996, these issues appear to have been corrected. The PCI BIOS Specification was published in 1994, but it was not until after Window 95 became widely deployed that BIOS writers get their act together and make these functions work.

The current FreeBSD implementation requires that the BIOS implement the BIOS32 interface. As such, some older machines will not allow interrupts to be routed, and must fall back to using tradition ISA interrupt routing (if possible). Like Windows 98 and newer, it requires that the BIOS be at least PCI 2.1-compliant and provides the $PIR Interrupt routing Table.

6.2 Bad PCI config space

The PCI Standard[PCI] requires that all devices that can have interrupts routed to them, but do not currently, have an INTLINE (Configuration Register offset 0x3c) of 0xff. However, a large number of devices have been seen in the wild with a value of 0. For IBM AT compatible systems, this is an illegal value. There is no way to route IRQ 0 to anything except the programmable interval timer, sometimes used to keep track of time.

Many PCI CardBus bridges either default to a value of 0 for INTLINE, or the BIOS on the machine that they are in writes a value of 0 into INTLINE. The PC Card subsystem in FreeBSD properly doesn't know about these problems. The PCI subsystem had to have code added to it to work around these buggy implementations.

6.3 Standards Deviation

The PC Card standard[PC Card] for PCI bridges (formerly known as YENTA) is incomplete. It does not specify a way, for example, to cause the card function interrupts to be routed using ISA interrupts. And there are a number of minor, but important, variations between vendors on how to do simple things. The author worked around the need to have varying code initialize and manage the bridges by using the fairly standard jump table technique. Each family of PCI-PC Card bridges had its own set of functions, and a table pointing to them. The rest of the FreeBSD PC Card system uses these tables to manage the bridge as needed.

On problem the author had in creating these functions and tables was gaining access to all the different families of devices. There are five major families of "YENTA" compliant PCI CardBus bridges. Cirrus Logic makes one family, Texas Instruments another, Toshiba a third, Ricoh a forth and O2Micro a fifth. The Texas Instrument ones are the most commonly used, and easiest to find and support. While the O2Micros were the hardest to find (the author still doesn't have a machine with an example bridge), they turned out to be the most standard and the generic routines easily supported them. Toshiba's ToPIC family has proven to be the hardest to support (as well as moderately difficult for the author to afford for laptops with the newer members of the ToPIC family).

6.4 3.3V 16-bit PC Cards

The "YENTA" specification spells out a uniform standard to supply voltages to PC Cards. This standard replaced the multitude of pre-existing methods to control card voltage. Prior to "YENTA" there were at least 5 different 3.3V standards: Two from Cirrus Logic, one from Intel, one from Ricoh, and one from Vadem. Since the original PCIC hardware didn't support 3.3V cards, each of these methods extended the original ExCA register set in different, incompatible ways.

One would like to use the "YENTA" interface to configure card voltages. However, many PCI bridges take using this interface to mean that the whole "YENTA" interface will be used to program the card, and where it replaces functionality of the old ExCA register set, that functionality will no longer work. The TI based chip-sets were found to break badly when using only the power interface.

This area is one area that would greatly benefit from further study. Debugging of the code which attempted to implement the "YENTA" power API was unable to turn up the cause of the difficulties. The author failed to make it work, and believes this would be a fruitful area of exploration.

6.5 Those Pesky Legacy Laptops

Before the "YENTA" specification was widely implemented, both Intel and Cirrus Logic produced PCI-PCMCIA bridges that looked more like a PCIC on PCI than "YENTA". PCIC is the name for the original Intel ISA to PCMCIA bridge chip (the 82365). These chip-sets required greater knowledge of how the interrupts are connected to the chip-set than "YENTA" chip-sets.

Cirrus Logic produced one PCI chip that followed this form. The CL-PD6729 glued a PCIC interface on the PCI bus. It used the same I/O index+data registers that the PCIC chip used. It used the Cirrus Logic variant of the ExCA register set, and could either be wired to deliver interrupts on the PCI bus, or wired to deliver interrupts on the ISA bus. There was no easy way for the driver writer to know which way the bridge was physically wired. The author assumed it was ISA, with the intention to make it possible for the user to specify PCI at a later date. This curiosity would have remained an obscure footnote to the computer industry had it not been extremely popular in Pentium 120, 133 and 150 based laptops. Surprisingly, many of these laptops are still in service today.

Intel also produced a similar PCI chip. The 82092AA implemented what its data-sheet calls the PPEC register set. This part has a PCMCIA bridge, as well as an IDE bridge on it. Only evaluation boards were ever made of this device. However, a number of clone makers cloned or licensed this design, and it appears in a few older laptops. Fortunately, its interface is similar enough to the CL-PD6729 that the same code works for its clones as well (and presumably on it, but the author has been unable to locate an existing PCI add-in card or laptop that uses this chip). The PPEC register set differs only in how it implements 3.3V extensions.

6.6 The Newest Laptops

The very newest laptops surprising showed a number of problems. The quality of their PCI BIOS seems to have started to vary widely. The older laptops tended to have good PCI BIOS support due to its mandate in the *PC 97 Design Guide*[PC97]. Older laptops with first generation "YENTA" parts on them had problems with their PCI BIOS implementation, which isn't surprising. It did surprise the author the number of new laptops that have this problem.

Months after the code was committed to "current," ACPI support was added to the tree. One of the many things ACPI does is to route PCI interrupts. The newer laptops that had been having problems using PCI BIOS to route the interrupts suddenly started working when ACPI was used to route the same interrupt. ACPI support has been mandated since the *PC 98 Design Guide*[PC98], so it is not surprising that this code path is tested more by systems integrators than the non-ACPI code path that the PCI BIOS takes. It is believed that the latest versions of Windows use ACPI more than PCI BIOS, so vendors are less likely to discover problems in the PCI BIOS API than they are in the ACPI.

7 Remaining problems

Three problems remain in the code. Most Toshiba ToPIC devices do not work when set to CardBus mode in the BIOS. Some systems will hang on the reboot process due to an interrupt storm. A small number of systems will hang on card insertion events, due to an interrupt storm. The author is aware of these problems, and is attempting to acquire hardware or access to hardware that exhibits these problems. Users experiencing other problems are invited to write to the author, or to the **mobile@freebsd.org** mailing list.

8 Development Model Weakness

Although not related to the hardware, or an existing flaw in the software, one final problem should be noted. FreeBSD's development model, which normally operates effectively failed to catch major problems in the PCI implementation before it was released into the stable tree. Fortunately, the process tends to correct itself, and these flaws didn't appear in 4.4-RELEASE.

FreeBSD keeps its source code in a CVS[CVS] tree. FreeBSD branches major releases in this tree, and then starts on its next major release. The branched trees are called "stable branches" because they aren't supposed to contain fully tested code. The main development branch is called "current" and contains code that might not be finished yet. Code is committed first to the "current" branch (yes, although it is technically the trunk, people often call it a branch). There people test it, fix problems with it and allow it to solidify. Once the code has been in "current" for a while, and appears ready for general users, and it is functionality needed in a prior branch, it is merged to "stable." In theory, this development mode ensures that no bad code is merged into "stable" since it has undergone testing in "current" first.

The author committed all of the changes to implement the functionality described in this paper to FreeBSD's "current" branch. He then prepared patches against "stable" that included his changes, as well as fairly extensive changes to the PCI layer to use PCI BIOS more aggressively. These changes had been in "current" for almost a year at this point. The original author of them had no knowledge of any major problems, except with a few off brand motherboards. The patches were posted to several mailing lists, and while problems were found in the PC Card code, none were reported in the PCI merge I was doing.

Within 48 hours of my committing the highly tested patch set (which itself was tested in current for a long time), I had to back out a large part of the PCI patch. It turns out that a number of PCI BIOS implementations do not report all the PCI devices on a bus; filters the config space in unexpected ways; and sometimes would hang the system unexpectedly. The extent and magnitude of these problems took the author of the PCI code and myself by surprise. Those two days showed that the convergence testing that we thought had been happening in "current" actually was minimal and insufficient for finding these problems.

Within a week of the merge into "stable," about 15 different bugs were reported (and mostly fixed) in

the PC Card code I had merged. Many of these bugs were people using less popular PCI-PC Card bridges with the new code. Some were configuration errors, or bad configurations that the FreeBSD PC Card layer could have detected but didn't. While most of these problems were easy to work through, some exist to this day. As large a user base that I had testing these patches before I did the merge was still none the less insufficient to ensure high quality for everyone.

While not directly related to the hardware quirks of using PCI interrupt routing, it does show that code dealing with both the generic PCI bus as well as the PC Card bus must be tested on a huge range of hardware, and be prepared to cope with a larger than expected number of unconforming BIOS and hardware implementations. The number of problem machines, and the nature of their problems surprised the author, who has been committing kernel code to FreeBSD for several years.

9 Acknowledgments

The author would like to thank Monzoon Networking, LLC for funding the development of this software. The author would also like to thank the dozens of testers that helped him debug preliminary versions of this software.

10 Availability

The software described in this paper have been integrated into the FreeBSD 4.4-RELEASE. FreeBSD is available free of charge for download from **http://www.freebsd.org/**.

References

[CVS] **http://www.cvshome.org/**

[FreeBSD] **http://www.freebsd.org/**

[Linux-CS] **http://pcmcia-cs.sourceforge.net/**

[NetBSD] **http://www.netbsd.org/**

[PC97] *PC 97 System Design Guide.* Microsoft Corporation, (1996).
http://www.pcdesguide.org /download/pc97.zip

[PC98] *PC 98 System Design Guide.* Intel Corporation and Microsoft Corporation, (1997).
http://www.intel.com/design /pc98/draft/pc98.pdf

[PC99] *PC 99 System Design Guide.* Intel Corporation and Microsoft Corporation, (1998).
http://www.pcdesguide.org /pc99/default.htm

[PC99a] *PC 99A Addendum.* Intel Corporation and Microsoft Corporation, (1999).

[PC2001] em PC 2001 System Design Guide, A Technical Reference for Designing PCs and Peripherals for the Microsoft Windows Family of Operating Systems. Intel Corporation and Microsoft Corporation, (2000).
http://www.pcdesguide.org /pc2001/default.htm

[PC Card] *PC Card Standard, Release 7.0*, PCMCIA, (February 1999).
http://www.pcmcia.org

[PC-98] **http://www.pc98.nec.co.jp/**

[PCI] *PCI Local Bus Specification, Revision 2.2*, PCI Special Interest Group, (December 18, 1998).
http://www.pcisig.com

[PCI BIOS] *PCI BIOS SPECIFICATION, Revsion 2.1*, PCI Special Interest Group, (August 26, 1994).

[WiFi] **http://www.wi-fi.org/**

Experiences on an Open Source Translation Effort in Japan

Hiroki Sato Keitaro Sekine
Faculty of Science and Technology, Tokyo University of Science, JAPAN
FreeBSD Japanese Documentation Project
hrs@jp.FreeBSD.org

Abstract

As network connectivity becomes more world-wide, the importance of translation efforts among open source software projects has grown rapidly. Many projects, including FreeBSD, NetBSD, and OpenBSD, already have some teams responsible for translating documents into other languages, but there seems to be few reports in respect to problems, efficiency, and so forth around the projects.

Translation work has several common characteristics and problems that are obviously different from typical software development. This paper describes them through my experiences with FreeBSD Japanese Documentation Project (doc-jp) for last two years, and points out the tasks that are required for such work.

1 Introduction

In countries where English is not the mother tongue, many people mainly get information about open source software via translated documents and have development activities in their own language. This is because of the fact that a lot of open source projects use English as the common language. As open source projects extended, their translation work [1–3] is as important for people as their English documentation efforts are.

In many cases, the translation efforts are not enough, since they need skilled people who understand at least two languages and know the software itself and some problems specific to translation. Things such as security advisories, which must be translated and spread all over the world as quickly as possible can make translation work more difficult than one had expected and can result in undesirable situations.

For this reason, we should consider efficient methodology on such translation work as part of a software development project. Through this paper, I will discuss several common characteristics and problems of the translation work that are obviously different from software development through my experience.

2 FreeBSD Japanese Documentation Project

2.1 History and Overview

To begin with, I would like to introduce a translation project which I belong to, which is the FreeBSD Japanese Documentation Project [1] (doc-jp). The project was started by some FreeBSD developers in Japan in 1995, and it plays an active part in translation of FreeBSD-related documents from FreeBSD Documentation Project [4] (FDP) into Japanese and the management other Japanese documents. Another project, the FreeBSD Japanese Manual Project [5] (JPMAN) is responsible for translating FreeBSD manual pages into Japanese. JPMAN and doc-jp complement each other, and are two primary translation efforts of FreeBSD in Japan.

The results of doc-jp's translation work over five years are as follows:

- FreeBSD Handbook (1996-, maintenance phase)
- FreeBSD FAQ (1997-, maintenance phase)
- FreeBSD Tutorials (1997-, maintenance phase)
- www.FreeBSD.org (1997-, maintenance phase)
- FreeBSD Release Notes/Errata (1997-, per release basis)

- FreeBSD Security Advisory (2000-, per release basis)
- Online version of "The design and implementation of 4.4BSD Operating System, Chapter 2" (2001, maintenance phase)
- FreeBSD-announce (important announcements only)

Most of the above are now actively maintained. The classifications of the work phases are described later.

The basic work of doc-jp is quite simple. A translator fetches a document in English via CVS, and then translates and submits it. Other project members review the translation and discuss it as the need arises. These processes are mainly dealt with on the doc-jp mailing list. When finished, a FreeBSD committer commits the translation into the FreeBSD CVS repository.

3 Characteristics of the Work

As described above, the process of translation work is similar to typical software development in a sense, but it seems that there are remarkable differences among them. In this section, several characteristics specific to translation that I noticed through doc-jp's work are described.

3.1 Review and Quality Evaluation

As you know, computer software runs on a computer, so the primary quality and functionality of code are determined objectively by their behavior themselves. Quality and functionality of translation, however, are obtained only by the review of project members and readers. Unfortunately, there are few software tools that can help us.

There are various kinds of translation "bugs" (e.g., typos, mistranslations, and so on), and to translate technical terms that are not familiar into Japanese requires "good taste" in translation efforts since it is subjective and sometimes leads to a conflict of members' opinions, which in turn can make translation more difficult than that of a programming project.

In short, the work not only needs good translators, but also good reviewers. This problem can affect a small project more than a larger one because the number of bilingual people involved is most likely less than in a large project.

3.2 The Life Span of Translated Documents

Another problem that can arise is the fact that original documents can be revised. Naturally, old code is also improved upon, but we can still use the old one if no fatal bugs exist. Older translated documents, on the other hand, usually cannot be used since in most cases the original ones are revised when they are obsolete or no longer reflect reality.

The general belief is that he translation work is to translate existing English documents into another language. While this is certainly true, it is even more important to maintain the documents after translation. Translated documents that disagree with the originals are of a negative value only, so the translators must avoid such situation. And, as with any open source project, keeping project members' motivated is extremely essential. The translation of documents is very hard work. Frankly speaking, maintenance of translated documents is not very creative for people that can understand the English document. After all, translated documents are no different than the original English documents.

3.3 Classification of Translation Phases

A translation work can be classified into the following three phases; first translation phase, maintenance phase, and feedback phase.

The first phase is to translate an English document into Japanese. This is called "the first translation phase." Once translators are collected, their motivation is relatively high. Work usually goes well in this phase until a first draft is submitted by the translators.

The second phase, called the "maintenance phase," is when submitted translations are reviewed and refined. This is the hardest and the longest job of the three.

The third phase is "feedback phase." When the translated documents are revised enough and become stable, frustration to fix bugs in the original documents arise by degrees. It is more or less worth revising original documents that are difficult to understand for translators, so the work could be effective as "good review" for the original ones.

When this classification is applied, most of the translated documents that already exist are in the "maintenance phase." Again, it should be emphasized that translation work is not temporary work. "Review and refine" in the second phase have the most relative importance throughout the work, making translation work as much of a continuous effort as the software development.

3.4 Statistics

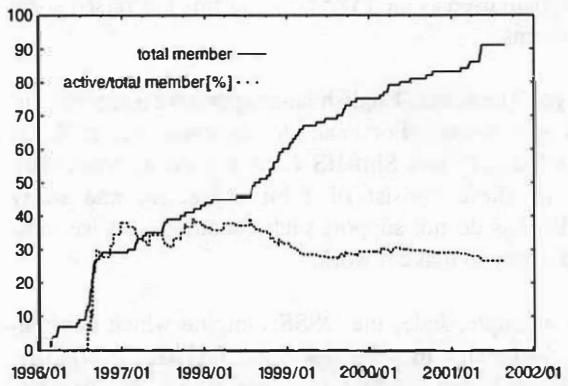

Figure 1: The growing number of doc-jp's members in proportion to active members.

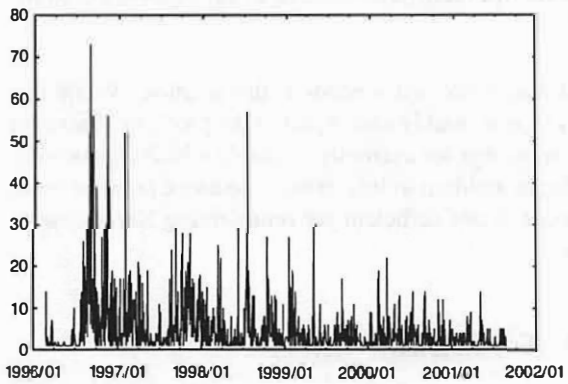

Figure 2: Distribution of the amount of mail on the doc-jp mailing list.

Figure 1 shows the growing number of doc-jp members in proportion to active members (defined here as a member who has posted more than 50 times to the doc-jp mailing list). Figure 2 shows the amount of email, on the doc-jp mailing list over a span of the last few years. Figure 3 shows the distribution of the translation phases for a particular document.

As shown in the figures, the member is growing, but the number of active persons still is few. A few of the members have continued to work regularly over a long period of time. Most of their interests concentrate on the first phase, yet the second phase has the most relative importance of the three in respect to the amount of work. Contrary to what you may think, the first phase takes a relatively short amount of time when compared to the other two.

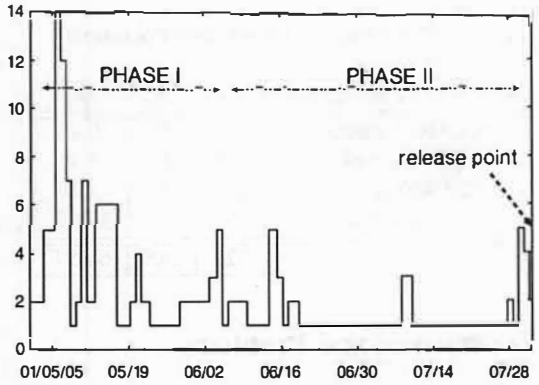

Figure 3: Distribution of the number of mail and translation phases for a certain document.

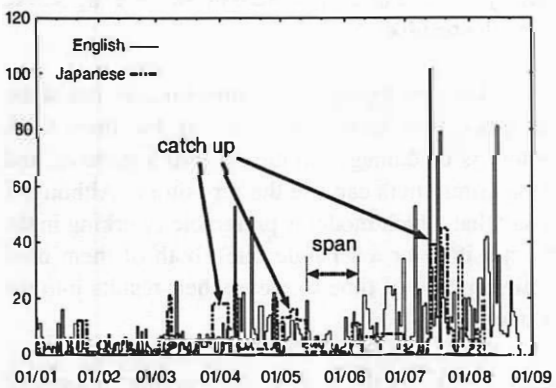

Figure 4: Commit frequency to FreeBSD doc/ and www/ tree.

Lastly, Figure 4 shows the commit frequency to the FreeBSD CVS repository's doc and www trees that FDP and doc-jp are responsible for. Notice that the FDP's work progresses regularly, but doc-jp's work is relatively intermittent. This is because the active members make the translated documents catch up with the originals when they are behind.

We have translated over 60% of the original documents in FreeBSD source tree into Japanese as of November 27, 2001, and have actively maintained them. A simple breakdown of them is shown in Table. 1.

Table 1: Percentages of translated documents.

location of the source tree	en	ja	%
doc/${LANG}/articles	26	8	30.8%
doc/${LANG}/books	72	47	65.3%
www/${LANG}	203	153	75.4%
src/release/doc/${LANG}	22	7	31.8%
Total	323	215	66.6%

4 Experiences and Problems

In this section, I will show several problems raised around the work. They may be relatively biased, but they are from real experiences.

4.1 Working Style and Release Engineering

As far as I know, some translation teams have their own CVS repositories. Doc-jp had its own repository in the past, but merged everything into the FreeBSD CVS repository three years ago because of the time lag before merging the results.

This doubled developing model undoubtedly has some advantages. Two such advantages are the main CVS repository is no damaged in the event of a mistake, and that non-committers can use the repository. Although I can't say that which model is preferable (working in the main repository or a separate one), both of them need a certain amount of time to merge their results into the main tree.

Roughly speaking, the release engineering process of FreeBSD is as follows. First the source tree is "frozen" and only the release engineers can make changes during the freeze period. This process usually lasts a few weeks to a month to work out any fatal bugs, and then it will be released. During this process, documentation is also prepared in parallel, however most release-related documents, such as release notes, etc., are prepared just before the release point, so we are usually pressed for time to translate them.

You might think that the translation should not be included with the release, however most translation teams would like to see the release documentation in their native language and have the documentation included. Under some circumstances, the translation cannot be prepared in time due to the lack of members and the lack of time available. As shown in Figure 3 and Figure 4, the translation work usually needs at least one or two weeks to catch up with the original documents.

4.2 Toolchain

Today, most of the documents in the FreeBSD source tree are marked up as DocBook/SGML or XML, so in order to make them readable, we need a toolchain to process them appropriately. Naturally, doc-jp uses the same toolchain used as the FDP, however this has raised some problems.

As you know, non-English languages have a specific encoding scheme. For example, Japanese has EUC-JP, ISO-2022-JP, and ShiftJIS (also known as MSKanji). All of these consist of 8-bit characters and many toolchains do not support such encodings, so we must find a way to make it work.

For example, Jade, the DSSSL engine which can output documents in several formats (HTML, PostScript, PDF, etc.) can produce Japanese output, but its TeX-based backend which is used for Postscript and PDF output, does not work properly. I am working on this issue, however since it is not completely, a PostScript or PDF version of FreeBSD Handbook in Japanese is not available.

You may think that unicode is the solution. While this may be true, it only solves part of the problem. There are few tools that are currently available which support unicode. In addition to this, from a Japanese point of view, unicode is not sufficient for representing Kanji characters.

4.3 For English Writers

This section discusses the kind of sentences and CVS operations that translators have the most trouble dealing with during translation efforts.

Figuratively speaking

While good sentences often include a figure of speech, translators tends to run into trouble when dealing with such complicated expressions. Most translators always expect sentences to be simple, straightforward, and to the point without going about so in a roundabout way.

A typical example is using a joke. Do you know of any jokes that people across the world can understand? It is extremely difficult to translate jokes into non-English languages and retain the humor.

Another example is slang, which does not appear in a dictionary. They also make translation work very difficult. I never insist that jokes should be kept out of documents, however, they should be kept conservative for good understanding.

It is also preferable to write full sentences and not one or two word phrases. Some experienced translators can understand such expressions, but it tends to mislead the translators. Remember, simple is preferable.

Unfortunately, I cannot give many examples because the reader would need knowledge of both Japanese and English in order to understand them, but I ask that you remember this; if your message is valuable, it can be translated even if it is a post to a mailing list or newsgroup. In order to increase the chances to find folks in other countries interested in your message, use sentences which they can translate smoothly.

The Rule of "Separate Commits"

In CVS, original documents should follow the rule of "carefully separated commits." This means that any commit to original documents should be divided into cosmetic changes and content changes. If the two are not divided properly, the diff deltas generated by CVS grow unnecessarily large. Most cosmetic changes have nothing to do with translation, so translation teams always appreciate separate commits because it reduces the amount of the work needed placed upon them.

For example, consider there is a SGML document that consists of two paragraph enclosed with <para> as shown below.

```
<para>This is a sample document marked up
  as DocBook/SGML.  If you are familiar
  with HTML, understanding SGML documents
  is not difficult.</para>

<para>Now, consider what kinds of
  difficulties there are in management
  of SGML documents.</para>
```

When a commit is made that changes the spacing of the first paragraph, the document and delta generated by CVS could look like this:

The modified document:

```
<para>This is a sample document marked up
  as DocBook/SGML.
  If you are familiar with HTML,
  understanding SGML documents
  is not difficult.</para>

<para>Now, consider what kinds of
  difficulties there are in management
  of SGML documents.</para>
```

The delta:

```
 <para>This is a sample document marked up
-  as DocBook/SGML.  If you are familiar
-  with HTML, understanding SGML documents
+  as DocBook/SGML.
+  If you are familiar with HTML,
+  understanding SGML documents
   is not difficult.</para>
```

If the "separated commits" rule is not followed, the translator should carefully compare deltas like those seen above. This is a very difficult and wasteful effort. A simple sign such as "cosmetic changes only" in the CVS log and a carefully separated commit greatly helps us reduce the amount of the work in translation.

The rule of "separate commits" originated in the FDP for this reason. Recently, I have noticed another situation that gives translators trouble. The following is the same two sentences as in the above example, however, they are interchanged. This often occurs when documents are being re-organized.

```
<para>Now, consider what kinds of
  difficulties there are in management
  of SGML documents.</para>

<para>This is a sample document marked up
  as DocBook/SGML.  If you are familiar
  with HTML, understanding SGML documents
  is not difficult.</para>
```

This change generates the following delta:

```
+<para>Now, consider what kinds of
+  difficulties there are in management
+  of SGML documents.</para>
+
 <para>This is a sample document marked up
    as DocBook/SGML.  If you are familiar
    with HTML, understanding SGML documents
    is not difficult.</para>
-
-<para>Now, consider what kinds of
-  difficulties there are in management
-  of SGML documents.</para>
```

While this does not seem to be a widely known fact, the interchanging of sentences can confuse translators very much. It increases the size of the delta that the translators must examine, and can very difficult to understand. You can imagine such interchange occurs in more complicated way.

I suggest that the changes described above should be considered cosmetic changes and separately committed from content changes. CVS logs help very much in these situations, so when such a change is made, please take what sort of change it is into account when writing the CVS commit message.

In short, as translators, we hope that those writing documentation will pay more attention to the translation work. If this is done, the documents will be able to be read by a larger amount of people, which is also good for the project.

5 For Efficiency

Up to this point, I described problems involving the both the translation work and the parent project. Next, I will show several ways for efficient translation work itself.

5.1 Providing Text Fragments But Whole

It is difficult for all of the translation project members to determine the status of the original and translated documents, so sometimes they hesitate over which to choose. If split documents are provided positively for the project's mailing list and so on, they can translate and review them immediately without unnecessary trouble.

Original documents that will be translated should be divided into relatively small text fragments and provided to translation project members. In addition, it is better for reviewers to keep the translated and the reviewed document side by side with the original text so they can easily compare the two.

5.2 Infrastructure

In doc-jp, translators need to fetch a target document via CVS, but doing so is sometimes difficult if the translator has no experience with CVS. Thus, an interface supporting translators with target documents and translated documents to be reviewed should be prepared. For instance, there is an experimental one for doc-jp [6], and other projects have similar facilities [7–9]. In particular, [7] is more functional since it includes reservation of translation.

Finally, older documents already translated should be marked so that people who read them are aware of their status. As mentioned earlier, obsolete documents are nothing but harmful for everyone. In doc-jp, a revision checker [10] is used for build process of the translated documents.

The revision check mechanism realized by [10] is quite simple. Original documents surely have a line of CVS ID like this (actually this is one line):

```
$FreeBSD:
doc/en_US.ISO8859-1/books/bandbook/book.sgml,v  1.119
2001/11/19 11:38:45 murray Exp $
```

And we make the translated documents have a line indicating its parent document as shown below:

```
Original revision: 1.119
$FreeBSD:
doc/ja_JP.eucJP/books/handbook/book.sgml,v  1.70
2001/10/27 18:12:06 hrs Exp $
```

The revision checker compares the CVS ID of the original document with the "Original revision" line in the translated document and the result is reflected in the definition of an entity called `%rev.diff;` as "IGNORE" or "INCLUDE" which used for a marked section of SGML. Actually, when the two revision is matched:

```
<!ENTITY % rev.diff 'IGNORE'>
...
<![ %rev.diff; [ this document is obsoleted! ]]>
```

and when they are not matched:

```
<!ENTITY % rev.diff 'INCLUDE'>
...
<![ %rev.diff; [ this document is obsoleted! ]]>
```

When the documents are rebuilt, this definition is included into each documents, so the documents themselves can notify the reader and maintainer that the translation is not up-to-date.

The important things are: 1) keep translated documents as up-to-date as possible, and 2) if circumstances do not allow this, notify the readers that the translated document is not up-to-date. The simple revision checker described above does just that.

5.3 Word list

During translation, we often think—especially when it is one of the technical terms—"what does this word mean?" To make things easy, we maintain a translation word list. Generally, it is a list which includes original and translated words on a word-by-word basis. Personally, I think there are problems with this and it is not sufficient.

First, maintenance of the list is relatively hard work. While many people translate documents, how do we determine which words should be candidates for the list? We have to discuss it, and the discussion usually takes quite a bit of time. Moreover, the objective answer is not always obtained.

Second, translation of sentences always goes with the sentence's context. The list of translation words does not include the context, it is possible to mislead the translators.

However, it is also undoubted that the words list is useful to keep consistency of translated words. The primary disadvantage is that it increases the project's work, and our goal is not to make a comprehensive word list.

I am designing an alternative that will identify already translated documents and includes a full-text search engine. Although it is not finished at the time of writing, using this method will allow translators to find the word they are looking for and the output will include a translated example sentence. Since the results of translation work are used as a database, I believe that the trouble described above can be somewhat relieved.

6 Summary and Future Directions

In this report, characteristics and problems specific to translation work are described through my experiences. To think little of translation efforts or to regard it as normal software development is the wrong idea.

I think that the translation efforts in various projects need much more technical cooperation and information exchange about their efficient management. The majority of frameworks can be shared, and the maintenance of them, such as word lists and style guides, can be done cooperatively instead of reinventing the wheel. The primary objective of the work is translation and not providing infrastructure itself.

With such a goal in mind, I have made a proposal for a project called the "Doc-ja Archive Project [11]," which supports various Japanese translation projects in early 2001. Unfortunately, the project virtually has obtained no results thus far, but we hope to become a place for discussion of translation efforts in Japan.

7 Acknowledgments

I thank Japan FreeBSD Users Group and FreeBSD Japanese Documentation Project for supporting my translation activities.

References

[1] FreeBSD Japanese Documentation Project, *"FreeBSD doc-jp web page,"* 1999
http://www.jp.FreeBSD.org/doc-jp/

[2] www.NetBSD.ORG Japanese Translation Project *"www.NetBSD.ORG Japanese Translation Project web page,"* 1999
http://www.jp.netbsd.org/ja/JP/Project/www-ja/

[3] www.OpenBSD.org Japanese Translation Project *"www.OpenBSD.org Japanese Translation Project web page,"* 2000
http://ja.open.4bsd.org/

[4] FreeBSD Documentation Project, *"FreeBSD Documentation Project Primer,"* 1999
http://www.FreeBSD.org/docs/en_US.ISO8859-1/fdp-primer/

[5] FreeBSD Japanese Manual Project, *"FreeBSD JPMAN web page,"* 1997
http://home.jp.freebsd.org/man-jp/

[6] FreeBSD Japanese Documentation Project, *"Synchronization status for FreeBSD Documentation Project,"* 2000
http://www.jp.FreeBSD.org/doc-jp/syncstat/

[7] FreeBSD Japanese Manual Project, *"JPMAN web reservation system,"* 1997
http://home.jp.freebsd.org/man-jp/yoyaku/

[8] Linux Japanese FAQ (JF) Project, *"JF in progress,"* 1999
http://www.linux.or.jp/JF/workshop/JF-in-Progress.html

[9] Debian Description Translation Server Project, *"Announcement of Debian Description Translation Server being available (in Japanese),"* 2001
http://lists.debian.or.jp/debian-doc/200108/msg00006.html

[10] FreeBSD Japanese Documentation Project, *"A script for SGML preprocessing and revision checking,"* 2000
http://cvsweb.FreeBSD.org/www/ja/prehtml

[11] Doc-ja Archive Project, *"Doc-ja Archive Project web page,"* 2001
http://openlab.ring.gr.jp/doc-ja/

Locking in the Multithreaded FreeBSD Kernel

John H. Baldwin
The Weather Channel
jhb@FreeBSD.org, http://people.FreeBSD.org/~jhb

Abstract

About a year ago, the FreeBSD Project embarked on the ambitious task of multithreading its kernel. The primary goal of this project is to improve performance on multiprocessor (MP) systems by allowing concurrent access to the kernel while not drastically hurting performance on uniprocessor (UP) systems. As a result, the project has been dubbed the SMP next generation project, or SMPng for short.

Multithreading a BSD kernel is not just a one-time change; it changes the way that data integrity within the kernel is maintained. Thus, not only does the existing code need to be reworked, but new code must also use these different methods. The purpose of this paper is to aid kernel programmers in using these methods.

It is assumed that the audience is familiar with the data integrity methods used in the traditional BSD kernel. The paper will open with a brief overview of these traditional methods. Next, it will describe the synchronization primitives new to the multithreaded FreeBSD kernel including a set of guidelines concerning their use. Finally, the paper will describe the tools provided to assist developers in using these synchronization primitives properly.

1 Introduction

Prior to the SMPng project, SMP support in FreeBSD was limited to the i386 architecture and used one giant spin lock for the entire kernel as described in Section 10.2 of [Schimmel94]. This kernel architecture is referred to as pre-SMPng. The goal of SMPng is to allow multiple threads to execute in the kernel concurrently on SMP systems.

2 Current Status

The SMPng project first began in June of 2000 with a presentation given to several FreeBSD developers by Chuck Paterson of BSDi explaining BSDi's SMP project to multithread their own kernel. This meeting set the basic design, and developers started working on code shortly after.

The first step was to implement the synchronization primitives. Once this was done, two mutexes were added. A spin mutex named `sched_lock` was used to protect the scheduler queues, sleep queues, and other scheduler data structures. A sleep mutex named `Giant` was added to protect everything else in the kernel. The second step was to move most interrupt handlers into interrupt threads. Interrupt threads are necessary so that an interrupt handler has a context in which to block on a lock without blocking an unrelated top half kernel thread. A few interrupt handlers such as those for clock interrupts and serial I/O devices still run in the context of the thread they interrupt and thus do not have a context in which to block. Once this was done, the old `spl` synchronization primitives were no longer needed and could be converted into nops. They are still left around for reference until code is converted to use locks, however.

Now that most of the infrastructure is in place, the current efforts are directed at adding locks to various data structures so that portions of code can be taken out from under `Giant`. There are still some infrastructure changes that need to be implemented as well. These include implementing a lock profiler that can determine which locks are heavily contested as well as where they are heavily contested. This will allow developers to determine when locking needs to be made more fine-grained.

3 Problems Presented by Concurrent Access to the Kernel

Multiple threads of execution within a BSD kernel have always presented a problem. Data structures are generally manipulated in groups of operations. If two concurrent threads attempt to modify the same shared data structure, then they can corrupt that data structure. Similarly, if a thread is interrupted and another thread accesses or manipulates a data structure that the first thread was accessing or manipulating, corruption can result. Traditional BSD did not need to address the first case, so it simply had to manage the second case using the following three methods:

- Threads that are currently executing kernel code are not preempted by other threads,
- Interrupts are masked in areas of the kernel where an interrupt may access or modify data currently being accessed or modified, and
- Longer term locks are acquired by synchronizing on a lock variable via `sleep` and `wakeup`.

With the advent of MP systems, however, these methods are not sufficient to cover both problematic cases. Not allowing threads in the kernel to be preempted does nothing to prevent two threads on different CPUs from accessing the same shared data structure concurrently. Interrupt masking only affects the current CPU, thus an interrupt on one CPU could corrupt data structures being used on another CPU. The third method is not completely broken since the locks are sufficient to protect the data they protected originally. However, race conditions on the locking and unlocking thread itself can lead to temporary hangs of threads. For a more detailed explanation see Chapter 8 of [Schimmel94] and Section 7.2 of [Vahalia96]. For an explanation of how these protections were implemented in 4.4BSD and derivatives see [McKusick96].

The pre-SMPng kernel addressed this situation on SMP systems by only allowing one processor to execute in the kernel at a time. This preserved the UP model for the kernel at the expense of disallowing any concurrent access to the kernel.

4 Basic Tools

Fortunately, MP-capable CPUs provide two mechanisms to deal with these problems: atomic operations and memory barriers.

4.1 Atomic Operations

An atomic operation is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and whose operation is safe from interference by other CPUs. For example, reading or writing a word of memory is an atomic operation. Unfortunately, reading and writing are only of limited usefulness alone as atomic operations. The most useful atomic operations allow modifying a value by both reading the value, modifying it, and writing it as a single atomic change. The details of FreeBSD's atomic operation API can be found in the atomic manual page [Atomic]. A more detailed explanation of how atomic operations work can be found in Section 8.3 of [Schimmel94].

Atomic operations alone are not very useful. An atomic operation can only modify one variable. If one needs to read a variable and then make a decision based on the value of that variable, the value may change after the read, thus rendering the decision invalid. For this reason, atomic operations are best used as building blocks for higher level synchronization primitives or for noncritical statistics.

4.2 Memory Barriers

Many modern CPUs include the ability to reorder instruction streams to increase performance [Intel00, Schimmel94, Mauro01]. On a UP machine, the CPU still operates correctly so long as dependencies are satisfied by either extra logic on the CPU or hints in the instruction stream. On a SMP machine, other CPUs may be operating under different dependencies, thus the data they see may be incorrect. The solution is to use memory barriers to control the order in which memory is accessed. This can be used to establish a common set of dependencies among all CPUs. An explanation of using store barriers in unlock operations can be found in Section 13.5 of [Schimmel94].

In FreeBSD, memory barriers are provided via the

atomic operations API. The API is modeled on the memory barriers provided on the IA64 CPU which are described in Section 4.4.7 of [Intel00]. The API include two types of barriers: acquire and release. An acquire barrier guarantees that the current atomic operation will complete before any following memory operations. This type of barrier is used when acquiring a lock to guarantee that the lock is acquired before any protected operations are performed. A release barrier guarantees that all preceding memory operations will be completed and the results visible before the current atomic operation completes. As a result, all protected operations will only occur while the lock is held. This allows a dependency to be established between a lock and the data it protects.

5 Synchronization Primitives

Several synchronization primitives have been introduced to aid in multithreading the kernel. These primitives are implemented by atomic operations and use appropriate memory barriers so that users of these primitives do not have to worry about doing it themselves. The primitives are very similar to those used in other operating systems including mutexes, condition variables, shared/exclusive locks, and semaphores.

5.1 Mutexes

The mutex primitive provides mutual exclusion for one or more data objects. Two versions of the mutex primitive are provided: spin mutexes and sleep mutexes.

Spin mutexes are a simple spin lock. If the lock is held by another thread when a thread tries to acquire it, the second thread will spin waiting for the lock to be released. Due to this spinning nature, a context switch cannot be performed while holding a spin mutex to avoid deadlocking in the case of a thread owning a spin lock not being executed on a CPU and all other CPUs spinning on that lock. An exception to this is the scheduler lock, which must be held during a context switch. As a special case, the ownership of the scheduler lock is passed from the thread being switched out to the thread being switched in to satisfy this requirement while still protecting the scheduler data structures. Since the bottom half code that schedules threaded interrupts and runs non-threaded interrupt handlers also uses spin mutexes, spin mutexes must disable interrupts while they are held to prevent bottom half code from deadlocking against the top half code it is interrupting on the current CPU. Disabling interrupts while holding a spin lock has the unfortunate side effect of increasing interrupt latency.

To work around this, a second mutex primitive is provided that performs a context switch when a thread blocks on a mutex. This second type of mutex is dubbed a sleep mutex. Since a thread that contests on a sleep mutex blocks instead of spinning, it is not susceptible to the first type of deadlock with spin locks. Sleep mutexes cannot be used in bottom half code, so they do not need to disable interrupts while they are held to avoid the second type of deadlock with spin locks.

As with Solaris, when a thread blocks on a sleep mutex, it propagates its priority to the lock owner. Therefore, if a thread blocks on a sleep mutex and its priority is higher than the thread that currently owns the sleep mutex, the current owner will inherit the priority of the first thread. If the owner of the sleep mutex is blocked on another mutex, then the entire chain of threads will be traversed bumping the priority of any threads if needed until a runnable thread is found. This is to deal with the problem of priority inversion where a lower priority thread blocks a higher priority thread. By bumping the priority of the lower priority thread until it releases the lock the higher priority thread is blocked on, the kernel guarantees that the higher priority thread will get to run as soon as its priority allows.

These two types of mutexes are similar to the Solaris spin and adaptive mutexes. One difference from the Solaris API is that acquiring and releasing a spin mutex uses different functions than acquiring and releasing a sleep mutex. A difference with the Solaris implementation is that sleep mutexes are not adaptive. Details of the Solaris mutex API and implementation can be found in section 3.5 of [Mauro01].

5.2 Condition Variables

Condition variables provide a logical abstraction for blocking a thread while waiting for a condition.

Condition variables do not contain the actual condition to test, instead, one locks the appropriate mutex, tests the condition, and then blocks on the condition variable if the condition is not true. To prevent lost wakeups, the mutex is passed in as an interlock when waiting on a condition.

FreeBSD's condition variables use an API quite similar to those provided in Solaris. The only differences being the lack of a `cv_wait_sig_swap` and the addition of `cv_init` and `cv_destroy` constructors and destructors. The implementation also differs from Solaris in that the sleep queue is embedded in the condition variable itself instead of coming from the hashed pool of sleep queue's used by `sleep` and `wakeup`.

5.3 Shared/Exclusive Locks

Shared/Exclusive locks, also known as sx locks, provide simple reader/writer locks. As the name suggests, multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock. Also, if one thread holds an exclusive lock, no threads may hold a shared lock.

FreeBSD's sx locks have some limitations not present in other reader/writer lock implementations. First, a thread may not recursively acquire an exclusive lock. Secondly, sx locks do not implement any sort of priority propagation. Finally, although upgrades and downgrades of locks are implemented, they may not block. Instead, if an upgrade cannot succeed, it returns failure, and the programmer is required to explicitly drop its shared lock and acquire an exclusive lock. This design was intentional to prevent programmers from making false assumptions about a blocking upgrade function. Specifically, a blocking upgrade must potentially release its shared lock. Also, another thread may obtain an exclusive lock before a thread trying to perform an upgrade. For example, if two threads are performing an upgrade on a lock at the same time.

5.4 Semaphores

FreeBSD's semaphores are simple counting semaphores that use an API similar to that of POSIX.4 semaphores [Gallmeister95]. Since `sema_wait` and `sema_timedwait` can potentially block, mutexes must not be held when these functions are called.

6 Guidelines

Simply knowing how a lock functions or what API it uses is not sufficient to understand how a lock should be used. To help guide kernel developers in using locks properly, several guidelines have been crafted. Some of these guidelines were provided by the BSD/OS developers while others are the results of the experiences of FreeBSD developers.

6.1 Special Rules About `Giant`

The `Giant` sleep mutex is a temporary mutex used to protect data structures in the kernel that are not fully protected by another lock during the SMPng transition. `Giant` has to not interfere with other locks that are added. Thus, `Giant` has some unique properties.

First, no lock is allowed to be held while acquiring `Giant`. This ensures that other locks can always be safely acquired whether or not `Giant` is held. This in turn allows subsystems that have their own locks to be called directly without `Giant` being held, and to be called by other subsystems that still require `Giant`.

Second, the `Giant` mutex is automatically released by the `sleep` and condition variable wait function families before blocking and reacquired when a thread resumes. Since the `Giant` mutex is reacquired before the interlock lock and no other mutexes may be held while blocked, this does not result in any lock order reversals due to the first property. There are lock order reversals between the `Giant` mutex and locks held while a thread is blocked when the thread is resumed, but these reversals are not problematic since the `Giant` mutex is dropped when blocking on such locks.

6.2 Avoid Recursing on Exclusive Locks

A lock acquire is recursive if the thread trying to acquire the lock already holds the lock. If the lock is

recursive, then the acquire will succeed. A recursed lock must be released by the owning thread the same number of times it has been acquired before it is fully released.

When an exclusive lock is acquired, the holder usually assumes that it has exclusive access to the object the lock protects. Unfortunately, recursive locks can break this assumption in some cases. Suppose we have a function F1 that uses a recursive lock L to protect object O. If function F2 acquires lock L, modifies object O so that is in an inconsistent state, and calls F1, F1 will recursively acquire L1 and falsely assume that O is in a consistent state.

One way of preventing this bug from going undetected is to use a non-recursive lock. Lock assertions can be placed in internal functions to ensure that calling functions obtain the necessary locks. This allows one to develop a locking protocol whereby callers of certain functions must obtain locks before calling the functions This must be balanced, however, with a desire to not require users of a public interface to acquire locks private to the subsystem.

For example, suppose a certain subsystem has a function G that is a public function called from other functions outside of this subsystem. Suppose that G is also called by other functions within the subsystem. Now there is a tradeoff involved. If you use assertions to require a lock to be held when G is called, then functions from other subsystems have to be aware of the lock in question. If, on the other hand, you allow recursion to make the interface to the subsystem cleaner, you can potentially allow the problem described above.

A compromise is to use a recursive lock, but to limit the places where recursion is allowed. This can be done by asserting that an acquired lock is not recursed just after acquiring it in places where recursion is not needed. However, if a lock is widely used, then it may be difficult to ensure that all places that acquire the lock make the proper assertions and that all places that the lock may be recursively acquired are safe.

An example of the first method is used with the `psignal` function. This function posts a signal to a process. The process has to be locked by the per-process lock during this operation. Since `psignal` is called from several places even including a few device drivers, it was desirable to acquire the lock in `psignal` itself. However, in other places such as signaling a parent process with `SIGCHLD` during process exit, several operations need to be performed while holding the process lock. This resulted in recursing on the process lock. Due to the wide use of the process lock, it was determined that the lock should remain non-recursive. Thus, `psignal` asserts that a process being signaled is locked, and callers are required to lock the process explicitly.

Currently in FreeBSD, mutexes are not recursive by default. A mutex can be made recursive by passing a flag to `mtx_init`. Exclusive sx locks never allow recursion, but shared sx locks always allow recursion. If a thread attempts to recursively lock a non-recursive lock, the kernel will panic reporting the file and line number of the offending lock operation.

6.3 Avoid Holding Exclusive Locks for Long Periods of Time

Exclusive locks reduce concurrency and should be held for as short a time as possible. Since a thread can block for an indeterminate amount of time, it follows that exclusive locks should not be held by a blocked thread when possible. Locks that should be held by a blocked thread are protecting data structures already protected in the pre-SMPng kernel. Thus, only locks present prior to SMPng should be held by a blocked thread, but new locks should not be held while blocked. The existing locks should be sufficient to cover the cases when an object needs to be locked by a blocked thread. An example of this type of lock would be a lock associated with a file's contents.

Functions that block such as the `sleep` and `cv_wait` families of functions should not be called with any locks held. Exceptions to this rule are the `Giant` lock, the optional interlock lock passed to the function, and locks that can be held by a blocked thread. Since these functions only release the interlock once, they should not be called with the interlock recursively acquired.

Note that it is better to hold a lock for a short period of time when it is not needed than to drop the lock only to reacquire it. Otherwise, the thread may have to block when it tries to reacquire the lock. For example, suppose lock L protects two lists A and B and that the objects on the lists do not need locks since they only have one reference at a time. Then, a section of code may want to remove an object from

list A, set a flag in the object, and store the object on list B. The operation of setting the flag does not require a lock due to the nature of the object, thus the code could drop the lock around that operation. However, since the operations to drop and acquire the lock are more expensive than the operation to set a flag, it is better to lock L, remove an object from list A, set the flag in the object, put the object on list B, and then release the lock.

6.4 Use Sleep Mutexes Rather Than Spin Mutexes

As described earlier, spin mutexes block interrupts while they are held. Thus, holding spin mutexes can increase interrupt latency and should be avoided when possible. Sleep mutexes require a context in case they block, however, so sleep mutexes may not be used in interrupt handlers that do not run in a thread context. Instead, these handlers must use spin mutexes. Spin mutexes may also need to be used in non-interrupt contexts or in threaded interrupt handlers to protect data structures shared with non-threaded interrupt handlers. All other mutexes should be sleep mutexes to avoid increasing interrupt latency. Also note that since locking a sleep mutex may potentially block, a sleep mutex may not be acquired while holding a spin mutex. Unlocking a contested sleep mutex may result in switching to a higher priority thread that was waiting on the mutex. Since we cannot switch while holding a spin mutex, this potential switch must be disabled by passing the `MTX_NOSWITCH` flag when releasing a sleep mutex while holding a spin mutex.

6.5 Lock Both Reads and Writes

Data objects protected by locks must be protected for both reads and writes. Specifically, if a data object needs to be locked when writing to the object, it also needs to be locked when reading from the object. However, a read lock does not have to be as strong as a write lock. A read lock simply needs to ensure that the data it is reading is coherent and up to date. Memory barriers within the locks guarantee that the data is not stale. All that remains for a read lock is that the lock block all writers until it is released. A write lock, on the other hand, must block all readers in addition to meeting the requirements of a read lock. Note that a write lock is always sufficient protection for reading.

Read locks and write locks can be implemented in several ways. If an object is protected by a mutex, then the mutex must be held for read and write locks. If an object is protected by an sx lock, then a shared lock is sufficient for reading, but an exclusive lock is required for writing. If an object is protected by multiple locks, then a read lock of at least one of the locks is sufficient for reading, but a write lock of all locks is required for writing.

An example of this method is the pointer to the parent process within a process structure. This pointer is protected by both the `proctree_lock` sx lock and the per-process mutex. Thus, to read the parent process pointer, one only needs to either grab a shared or exclusive lock of the `proctree_lock` or lock the process structure itself. However, to set the parent process pointer, both locks must be locked exclusively. Thus, the `inferior` function asserts that the `proctree_lock` is locked while it walks up the process tree seeing if a specified process is a child of the current process, while `psignal` simply needs to lock the child process to read the parent process pointer while posting `SIGCHLD` to the parent. However, when re-parenting a process, both the child process and the process tree must be exclusively locked.

7 Diagnostic Tools

FreeBSD provides two tools that can be used to ensure correct usage of locks. The tools are lock assertions and a lock order verifier named witness. To demonstrate these tools, we'll provide code examples from a kernel module [Crash] and then explain how the tools can be used. The module works by receiving events from a userland process via a sysctl. The event is then handed off to a kernel thread which performs the tasks associated with a specific event.

Both of these tools are only enabled if the kernel is compiled with support for them enabled. This allows a kernel tuned for performance to avoid the overhead of verifying the assertions while allowing for a debug kernel used in development to perform the extra checks. Currently both of these tools are enabled by default, but they will be disabled when the FreeBSD Project releases 5.0 for performance reasons. If either tool detects a fatal problem, then it will panic the kernel. If the kernel debugger is

compiled in, then the panic will drop into the kernel debugger as well. For non-fatal problems, the tool may drop into the kernel debugger if the debugger is enabled and the tool is configured to do so.

7.1 Lock Assertions

Both mutexes and shared/exclusive locks provide macros to assert that a lock is held at any given point. For mutexes, one can assert that the current thread either owns the lock or does not own the lock. If the thread does own the lock, one can also assert that the lock is either recursed or not recursed. For shared/exclusive locks, one can assert that a lock is either locked in either fashion. If an assertion fails, then the kernel will panic and drop into the kernel debugger if the debugger is enabled.

For example, in the crash kernel module, events 14 and 15 trigger false lock assertions. Event 14 asserts that `Giant` is owned when it is not.

```
case 14:
        mtx_assert(&Giant, MA_OWNED);
        break;
```

When event 14 is triggered, the output on the kernel console is:

```
crash: assert that Giant is locked
panic: mutex Giant not owned at crash.c:202
cpuid = 3; lapic.id = 03000000
Debugger("panic")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

Event 15 exclusively locks the sx lock `foo` and then asserts that it is share locked.

```
case 15:
        sx_xlock(&foo);
        sx_assert(&foo, SX_SLOCKED);
        sx_xunlock(&foo);
```

When event 15 is triggered, the output on the kernel console is:

```
crash: assert that foo is slocked
 while it is xlocked
panic: Lock (sx) foo exclusively locked
 @ crash.c:206.
cpuid = 1; lapic = 01000000
Debugger("panic")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

7.2 Witness

Along with the mutex code and advice provided by BSDi came a lock order verifier called witness. A lock order verifier checks the order in which locks are acquired against a specified lock order. If locks are acquired out of order, than the code in question may deadlock against other code which acquires locks in the proper order. For the purposes of lock order, a lock A is said to be acquired before lock B, if lock B is acquired while holding lock A.

Witness does not use a static lock order, instead it dynamically builds a tree of lock order relationships. It starts with explicit lock orders hard-coded in the source code. Once the system is up and running, witness monitors lock acquires and releases to dynamically add new order relationships and report violations of previously established lock orders. For example, if a thread acquires lock B while holding lock A, then witness will save the lock order relationship of A before B in its internal state. Later on if another thread attempts to acquire lock B while holding lock A, it will print a warning message on the console and optionally drop into the kernel debugger. The BSD/OS implementation only worked with mutexes, but the FreeBSD Project has extended it to work with shared/exclusive locks as well.

Locks are grouped into classes based on their names. Thus, witness will treat two different locks with the same name as the same. If two locks with the same name are acquired, witness will panic unless multiple acquires of locks of that name are explicitly allowed. This is because witness can not check the order in which locks of the same name are acquired. The only lock group that witness currently allows multiple acquires of member locks are process locks. This is safe because process locks follow a defined order of locking a child process before locking a parent process.

The crash kernel module was originally written to test witness on sx locks, thus it contains events to

violate lock orders to ensure that witness detects the reversals. Event 2 locks the sx lock `foo` followed by the the sx lock `bar`. Event 3 locks the sx lock `bar` followed by the sx lock `foo`.

```
case 2:
        sx_slock(&foo);
        sx_slock(&bar);
        sx_sunlock(&foo);
        sx_sunlock(&bar);
        break;
case 3:
        sx_slock(&bar);
        sx_slock(&foo);
        sx_sunlock(&foo);
        sx_sunlock(&bar);
        break;
```

When event 2 is triggered followed by event 3, the output on the kernel console is:

```
crash: foo then bar
crash: bar then foo
lock order reversal
 1st 0xc335fce0 bar @ crash.c:142
 2nd 0xc335fc80 foo @ crash.c:143
Debugger("witness_lock")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

Event 4 locks the sx lock `bar`, locks the sx lock `foo`, and then locks the sx lock `bar2`. The locks `bar` and `bar2` both have the same name and thus belong to the same lock group.

```
case 4:
        sx_slock(&bar);
        sx_slock(&foo);
        sx_slock(&bar2);
        sx_sunlock(&bar2);
        sx_sunlock(&foo);
        sx_sunlock(&bar);
        break;
```

When event 4 is triggered, the output on the kernel console is:

```
crash: bar then foo then bar
lock order reversa# 1
 1st 0xc3363ce0 bar @ crash.c:148
 2nd 0xc3363c80 foo @ crash.c:149
 3rd 0xc3363d40 bar @ crash.c:150
Debugger("witness_lock")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

Note that if there are actually three locks involved in a reversal, all three are displayed. However, if two locks are merely reversing a previously established order, only the information about the two locks is displayed.

In addition to performing checks, witness also adds a new command to the kernel debugger. The command `show locks [pid]` displays the locks held by a given thread. If a pid is not specified, then the locks held by the current thread are displayed. For example, after the panic in the previous example, the output is:

```
db> show locks
shared (sx) foo (0xc3363c80) locked
 @ crash.c:149
shared (sx) bar (0xc3363ce0) locked
 @ crash.c:148
```

Sleep mutexes and sx locks are displayed in the order they were acquired. If the thread is currently executing on a CPU, then any spin locks held by the current thread are displayed as well.

8 Conclusion

Multithreading a BSD kernel is not a trivial task. However, it is a feasible task given the proper tools and some guidelines to avoid the larger pitfalls. Volunteers seeking to work on the SMPng Project can check the SMPng Project web page [SMPng] for the todo list. The FreeBSD SMP mailing list (`freebsd-smp@FreeBSD.org`) is also available for those wishing to discuss issues with other developers.

9 Acknowledgments

Thanks to The Weather Channel; Wind River Systems, Inc.; and Berkeley Software Design, Inc. for

funding some of the SMPng development as well as this paper. Thanks also to BSDi for donating their SMP code from the BSD/OS 5.0 branch. Special thanks are due to those who helped review and critique this paper including Sam Leffler, Robert Watson, and Peter Wemm. Finally, thanks to all the contributors to the SMPng Project including Jake Burkholder, Matt Dillon, Tor Egge, Jason Evans, Brian Feldman, Andrew Gallatin, Greg Lehey, Jonathan Lemon, Bosko Milekic, Mark Murray, Chuck Paterson, Alfred Perlstein, Doug Rabson, Robert Watson, Andrew Reiter, Dag-Erling Smørgav, Seigo Tanimura, and Peter Wemm.

10 Availability

FreeBSD is an Open Source operating system licensed under the very liberal Berkeley-style license [FreeBSD]. It is freely available from a world-wide network of FTP servers. The SMPng work is being done in the development branch and is not presently in any released version of FreeBSD. It will debut in FreeBSD 5.0. Installation snapshots for the development versions of FreeBSD on the i386 architecture can be found at

```
ftp://snapshots.jp.FreeBSD.org/ \
    pub/FreeBSD/snapshots/i386
```

References

[Gallmeister95] Bill Gallmeister, *POSIX.4 Programming for the Real World*, O'Reilly & Associates, Inc. (1995) p. 134-146.

[Intel00] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: IA-64 Application Architecture*, Intel Corporation (2000).

[Mauro01] Jim Mauro and Richard McDougall, *Solaris Internals: Core Kernel Components*, Sun Microsystems Press (2001).

[McKusick96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Longman, Inc. (1996) p. 91-92.

[Schimmel94] Curt Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*, Addison-Wesley Publishing Company (1994).

[Vahalia96] Uresh Vahalia, *UNIX Internals: The New Frontiers*, Prentice-Hall, Inc. (1996).

[Atomic] *Atomic*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/ \ man.cgi?manpath=FreeBSD+5.0-current`

[Crash] Crash Example Kernel Module, `http://people.FreeBSD.org/~jhb/ \ crash/crash.c`

[FreeBSD] FreeBSD Project, `http://www.FreeBSD.org`

[SMPng] FreeBSD SMPng Project, `http://www.FreeBSD.org/smp`

Advanced Synchronization in Mac OS X: Extending Unix to SMP and Real-Time

Louis G. Gerbarg
Apple Computer, Inc.
louis@apple.com

Abstract

Throughout the years, as Unix has grown and evolved so has computer hardware. The 4.4BSD-Lite2 distribution had no support for two features that are becoming more and more important: SMP and real-time processing.

With the release Mac OS X Apple has made extensive alterations to our kernel in order to support both SMP and real-time processing. These alterations affected both the BSD and Mach portions of our kernel, as well as shaping our driver system, IOKit.

These changes range from scheduling policies, enabling support for kernel preemption, altering locking hierarchies, and defining new serialization primitives, as well as designing a driver architecture that allows developers to easily make their drivers SMP and preemption safe.

1 Introduction

Traditional BSD kernels do some things very well. SMP is not one of them. The 4.4BSD-Lite2 source, on which NetBSD, FreeBSD, OpenBSD, and Mac OS X are based did not have support for SMP. Its locking mechanisms were not set up for multiple processors, the kernel was not reentrant, and bottom half (interrupt time) drivers always work directly within an interrupt context. As FreeBSD, Mac OS X, and NetBSD have moved to support SMP they have had to overcome these shortcomings. Some aspects of their solutions are similar, some are wildly divergent. Both xnu and FreeBSD have decided to adopt interrupt thread contexts, as well as a number of similar new locking primitives.

2 An Introduction to xnu

Before delving into the intricacies of Mac OS X's advanced features a brief overview of the kernel architecture and its history is necessary. Mac OS X is based around a BSD distribution known as Darwin. At the heart of Darwin is its kernel, xnu. xnu is a monolithic kernel based on sources from the OSF/mk Mach Kernel, the BSD-Lite2 kernel source, as well as source that was developed at NeXT. All of this has been significantly modified by Apple.

xnu is not a traditional microkernel as its Mach heritage might imply. Over the years various people have tried methods of speeding up microkernels, including collocation (MkLinux), and optimized messaging mechanisms (L4)[microperf]. Since Mac OS X was not intended to work as a multi-server, and a crash of a BSD server was equivalent to a system crash from a user perspective the advantages of protecting Mach from BSD were negligible. Rather than simple collocation, message passing was short circuited by having BSD directly call Mach functions. While the abstractions are maintained within

the kernel at source level, the kernel is in fact monolithic. xnu exports both Mach 3.0 and BSD interfaces for userland applications to use. Use of the Mach interface is discouraged except for IPC, and if it is necessary to use a Mach API it should most likely be used indirectly through a system provided wrapper API.

3 Basic Synchronization

Operating systems use a number of structures and algorithms to ensure proper synchronization between various parts of the kernel. xnu uses several different locking structures, including the BSD lockmanager, Mach mutexes, simple locks, read-write locks, and funnels. Additionally thread control is complicated by the use of Mach continuations, and kernel preemption.

3.1 Simple Locks

Simple locks in Mach are standard spin locks. When a thread attempts to access a simple lock that is in use it loops until the lock becomes free. This is useful when allowing the thread to sleep could cause a deadlock, or when one of the threads could be running in an interrupt context.

Simple locks are the safest general synchronization primitive to use when in doubt, but their CPU cost is very high. In general is is better to use a mutex if at all possible. If a piece of code attempts to acquire a simple lock it already holds it will result in a kernel panic.

```
void
simple_lock_init(
    usimple_lock_t,etap_event_t);

void
simple_lock(usimple_lock_t);

void
simple_unlock(usimple_lock_t);

unsigned
int simple_lock_try(usimple_lock_t);
```

3.2 Mutexes

Mach mutexes are very primitive. Since they are sleep locks, and do not have the rich semantics that FreeBSDs mutexes have. They are sleep locks, when a thread attempts to access an inuse mutex it will sleep until that mutex is available. Mutexes can be used from a thread context (though it is not always the best performance decision for things like drivers). If a piece of code attempts to acquire a mutex it already holds it will result in a kernel panic.

```
//The etap_event parameter is
//deprecated, just pass a value of 0.
mutex_t *
mutex_alloc  (etap_event_t);

void
mutex_free (mutex_t*);

void
mutex_lock (mutex_t*);

void
mutex_unlock (mutex_t*);

boolean_t
mutex_try (mutex_t*);
```

3.3 Read-Write Locks

Many variables within the kernel are safe to be read, so long as they are not being written. If a lock is highly contended, generally it is primarily being protected for readers. Read-write locks solve this problem by allowing either multiple reads, or a single writer to possess the lock. While there are API's for promoting and demoting locks between the read and write states, their usage is discouraged and subject to change.

```
void
lock_write (lock_t*);

void
lock_read (lock_t*);

void
lock_done (lock_t*);

#define lock_read_done(l) \
    lock_done(l);
#define lock_write_done(l) \
    lock_done(l);
```

3.4 Continuations

One of the costs typically associated with context switches is saving and restoring thread stacks. This uses both CPU time and wired memory. In order to avoid this cost, Mac OS X uses Mach continuations whenever possible. A continuation allows the kernel to avoid saving or restoring a kernel stack across schedulings of the thread.

Continuations work within a non-preemptible context. Since the thread is not going to be preempted, its entry and exit points are well-defined. The thread begins executing through a call to a function pointer. It ends execution by making a call that tells the scheduler to schedule a new thread, and leaves a pointer to a function that should be executed the next time the thread is scheduled. It is the thread's responsibility to save and restore its own variables.

While it is useful to be aware of continuations, it is not generally necessary to directly interact with them. They may be useful for doing extremely low overhead threading, but in general it is best to use them indirectly through other kernel mechanisms such as IOWorkLoops.

```
void thread_set_cont_arg(int);
int thread_get_cont_arg(void);
```

4 Funnels: Serializing access to BSD

Funnels are quite possibly one of the most confusing elements of xnu for people familiar with other BSD kernels. They are not a lock in the traditional sense of the word (though they are sometimes referred to as "flock" within the kernel). Funnels are used to serialize access to the BSD segment of the kernel. This is necessary because that portion on the codebase does not have fine-grained locking, and is not fully reentrant. There are currently two funnels within the kernel, the kernel funnel (it might be more appropriate to call it the filesystem funnel, though it does protect a few calls besids the file systems), and the network funnel.

4.1 Funnels

Funnels first appeared into Digital UNIX[dgux], though their implementation in Mac OS X is entirely different, and significantly improved. Funnels are actually built on top of Mach mutexes. Each funnel backs into a mutex, and once a thread gains a funnel it is holding that funnel while it is executing. The difference between a funnel and a mutex is that a mutex is held across rescheduling. The scheduler drops a thread's funnel when it is rescheduled, and reacquires the funnel when it is rescheduled. That means that holding a funnel does not guarantee that another thread will not enter a critical section before a thread drops the funnel. What it does mean is that on a multiprocessor system it is guaranteed that no other thread will access the section concurrently from another CPU.

Originally there was a single funnel protecting the entirety of the BSD kernel. It was in many ways analoguous to FreeBSD-current's Giant mutex (more on that later). Since networking and other kernel functions are generally seperate, splitting the funnel into two is a major win for dual processor machines. Unfortunately, since holding

both funnels can result in nasty deadlocks and other problems, holding both at the same time causes a panic. This can cause significant problems for entities that need to access items that are protected by each funnel. The primary entities this effects are network file systems. The funnel API has a call for swapping funnels, but in some cases this has proven to be too complicated to orchestrate (such as NFS serving). The API also provides a merge call which will combine the two funnels into a single funnel, backed by a single mutex. Unfortunately, the funnels cannot be unmerged, which causes a net performance loss.

The primary difference between Digital UNIX funnels and Mac OS X funnels are that on Digital UNIX there can only be one funnel, and it always will be on the primary CPU. On Mac OS X there can be multiple funnels, and funnels can run on any CPU (although a particular funnel may only be on one CPU at any given time).

There are primitives for creating funnels, but in general nobody should be creating new funnels. All control of the funnels is done through the thread_funnel_set call().

```
boolean_t
thread_funnel_set
    (funnel_t * fnl, boolean_t funneled);
```

4.2 So long spl...

In BSD the various spl priority levels formed a locking hierarchy that could be used to guarantee synchronization between the interrupt and non-interrupt segments of a driver. Unfortunately the spl's definitions got less and less fine-grained over the years, and they were never particularly well suited for SMP. For these reasons Mac OS X no longer uses them. Instead it manages its synchronization through mutexes, and the BSD funnel serializations.

If this sounds familiar to FreeBSD users, that is probably because FreeBSD-current actually has a funnel (or rather a magic mutex), Giant. FreeBSD plays scheduler games with Giant that are almost identical to what xnu does with funnels, although Mac OS X deals with them explicitly, through a different API than its mutexes. Like FreeBSD, xnu has replaced the functionality of the spl's with these more flexible syncronization primitives. Unlike FreeBSD, the spl calls are still sprinkled through the kernel. Through the development of xnu they have been no-ops, wrappers to getting the funnels, and most recently they act as asserts to make sure the funnels are in the correct state when they are called.

5 Real-Time

There are two important aspects to real-time scheduling. One is the scheduling algorithm, the other is guaranteeing latencies within the kernel are not excessive. While both will be discussed, this section focuses mostly on the latencies related issues.

5.1 Interrupt Handling

True interrupt handlers cannot be preempted, and cannot sleep. Therefore, if there is a long path in an interrupt handler it will lead to high latency. In order to handle this, xnu generally uses a simple interrupt handler that processes the interrupt by triggering a handler in a regular kernel thread context that a driver has registered for the interrupt handler. This "pseudo interrupt" handler is run in a normal kernel thread context, where it can access the full kernel API. If true interrupt handling is necessary the correct mechanism is generally an IOFilterInterruptEventSource (see below).

5.2 Scheduling Bands

xnu internally has 128 priority levels, ranging from 0 (lowest priority) to 127 (highest priority). They are divided into several major bands. 0 through 51 correspond to what is available through the traditional BSD interface. The default priority is 31. 52 through 63 correspond to elevated priorities. 64-79 are the highest priority regular threads, and are used by things like WindowServer. 80 through 95 are for kernel mode threads. Finally 96 through 127 correspond to real-time threads, which are treated differently than other threads by the scheduler.

5.3 Fixed and Degrading priorities

By default the scheduler creates threads with degradable priorities. These threads will have lower and lower effective priorities as they use (and abuse) their time allocations. This is particularly significant for real-time threads, since if they are truly abusive they will eventually degrade into non-real-time threads. This mechanism means that it is possible to allow non-superusers to create real-time threads.

There are also mechanisms to create fixed priority threads which will not degrade. Their creation is much more restrictive than degradable threads, since they can be used very effectively to perform a denial of service against a system.

5.4 Kernel Preemption

Kernel preemption is the main tool xnu uses to achieve low latencies. The kernel is preemptible, though in standard usage kernel preemption is turned off. Kernel preemption begins when a real-time thread is scheduled. Since the real-time thread has a higher priority than a kernel thread it should be scheduled in favor of the kernel thread, and that is the point at which kernel preemption is activated.

Preemption changes the runtime characteristics of the kernel dramatically. Continuations are no longer nearly as useful, since the thread may be rescheduled at any point, which will require a stack. Additionally, all sorts of new deadlocks can arise. In order to cope with this the locking primitives have been modified to work with preemption. Simple locks disable preemption while they are spinning. Mutexes only disable preemption while the thread is trying to gain access to its interlock (a spin lock protecting the mutexes private data structures). Additionally the true interrupt handler is not preemptible

What this means is that well written code should not need to be at all aware of the fact that kernel preemption is enabled, and should just work if they properly use the locking primitives. It should be transparent to most kernel extensions and drivers. It may not be transparent if the driver uses an IOFilterInterruptEventSource, or does not make proper use of an IOWorkLoop, as described in the next section.

6 IOKit

IOKit is the driver subsystem of the Mac OS X kernel. IOKit provides a number of synchronization primitives, ranging from simple wrappers to the Mach primitives, all the way through complex new synchronization constructs that massively simplify writing drivers for devices that are SMP clean and preemptible. IOKit is implemented in eC++ [eC++], a subset of C++, and uses a custom runtime type system.

6.1 IOLocks

IOKit provides wrappers to the Mach locking primitives. These wrappers provide some convenience as well as a consistent interface to the locking primitives.

6.1.1 IORWLock

IORWLock provides a wrapper to the standard Mach read-write locks.

```
IORWLock *
IORWLockAlloc( void );

void
IORWLockFree( IORWLock * lock);

void
IORWLockRead( IORWLock * lock);

void
IORWLockWrite( IORWLock * lock);

void
IORWLockUnlock( IORWLock * lock);
```

6.1.2 IORecursiveLock

IORecursiveLock provides a wrapper to the standard Mach mutexes. Additionally, it has an internal reference counting mechanism that allows it to be locked recursively.

```
IORecursiveLock *
IORecursiveLockAlloc( void );

void
IORecursiveLockFree(
    IORecursiveLock * lock);

void
IORecursiveLockLock(
    IORecursiveLock * lock);

boolean_t
IORecursiveLockTryLock(
    IORecursiveLock * lock);

void
IORecursiveLockUnlock(
    IORecursiveLock * lock);
```

6.1.3 IOLock

IOLock provides a wrapper to the standard Mach mutexes. The semantics are the same. Recursive locking is not allowed.

```
IOLock *
IOLockAlloc( void );

void
IOLockFree( IOLock * lock);

void
IOLockLock( IOLock * lock);

boolean_t
IOLockTryLock( IOLock * lock);

void
IOLockUnlock( IOLock * lock);
```

6.1.4 IOSimpleLock

IOSimpleLock provides a wrapper to the standard Mach simple_locks. Additionally, it has an interface for enabling and disabling interrupts (drivers should probably be using a IOWorkLoop for synchronization, which will take care of interrupt related issues).

```
IOSimpleLock *
IOSimpleLockAlloc( void );

void
IOSimpleLockFree( IOSimpleLock * lock );

void
IOSimpleLockLock( IOSimpleLock * lock );

boolean_t
IOSimpleLockTryLock( IOSimpleLock * lock );

void
IOSimpleLockUnlock( IOSimpleLock * lock );

IOInterruptState
```

```
IOSimpleLockLockDisableInterrupt(
    IOSimpleLock * lock );

void
IOSimpleLockUnlockEnableInterrupt(
    IOSimpleLock * lock,
    IOInterruptState state );
```

6.2 IOWorkLoop

IOWorkLoops are constructs designed to simplify synchronization issues that arise when working with hardware in the multi-threaded, reentrant, preemptible environment present within xnu. Unlike the other locking primitives discussed earlier in this paper, the IOWorkLoop is a very complex entity that takes care of most of the more mundane synchronization issues for driver writers. Its interface is rather extensive, and somewhat complex.

The basic idea behind a work loop is that it forces anything attached to the work loop to run effectively single threaded. So while anything is holding the work loop none of the other event handlers or runActions associated can run. This effectively synchronizes the various items that are attached to the work loop. It also provides a convenient mechanism for servicing interrupts and timers while keeping them synchronized.

The work loop also takes care of a bunch of mundane issues such as turning on and off interrupts during certain locking procedures, meaning that driver writers can concentrate on getting their drivers working, not keeping their locking straight. Inherently there is some overhead in using work loops, and they do not serve every purpose, but they are quite flexible, and allow programmers to write correct drivers without intimate knowledge of xnu's internal synchronization mechanisms.

6.2.1 EventSources

IOEventSources are very flexible constructs for dealing with asynchronous events. While it is possible to implement new event sources, in general the provided IOInterruptEventSource and IOTimerEventSource are sufficient.

Event sources allow functions to be associated with asynchronous events, such as interrupts and timers. The full details and subtleties of how they work falls outside the scope of this paper, but the basic interfaces for creating new event sources are provided below.

Once an event source has been created it can then be added to a work loop. After that any time the event happens it will automatically be processed by the function that was specified when it was created, and in the work loop context.

```
IOTimerEventSource *
IOTimerEventSource::timerEventSource(
    OSObject *owner, Action action = 0);

IOInterruptEventSource *
IOInterruptEventSource::interruptEventSource(
    OSObject *owner, Action action,
    IOService *provider = 0,
    int intIndex = 0);

virtual IOReturn
IOWorkLoop::addEventSource(
    IOEventSource *newEvent);

virtual IOReturn
removeEventSource(
    IOEventSource *toRemove);
```

6.2.2 runActions

Event sources solve a significant amount of the synchronization issues drivers face dealing with the bottom half (interrupt time) of the driver, but they do not deal with the synchronizing the top half (non-interrupt)

and bottom half of the driver. This synchronization is achieved through the use of runActions.

runActions simply link a particular invocation of a function to the work loop. While the runAction is operating it is holding the work loop, thus forcing synchronization with everything else on the work loop, including the interrupt and timer event handlers.

```
typedef IOReturn (*Action)
    (OSObject *target,
    void *arg0, void *arg1,
    void *arg2, void *arg3);
```

```
virtual IOReturn
IOWorkLoop::runAction
    (Action action, OSObject *target,
    void *arg0 = 0, void *arg1 = 0,
    void *arg2 = 0, void *arg3 = 0);
```

6.3 IOFilterInterruptEventSource

IOFilterInterruptEventSource is a subclass of IOInterruptEventSource It is special because in addition to running within the work loop thread's context it runs directly on the primary interrupt context. This allows for much faster interrupt response time, but also means that an IOFilterInterruptEventSource cannot block, and must not use any kernel API that may block. In general IOFilterInterruptEventSoures should be used for cases where there are a lot of potential spurious interrupts, such as when a device shares an interrupt, or when processing only needs to be performed after several interrupts. The IOFilterInterruptSource can choose to ignore the interrupts that do not need processing, and pass the ones that do need processing onto an IOInterruptEventSource. A full description of limitations imposed on code running within the primary interrupt context is beyond the scope of this paper.

7 Conclusions

Darwin provides a number of synchronization primitives, both traditional and unique. They provide mechanisms for writing high performance drivers, without requiring driver writers to become intimately familiar with the OS. This both simplifies driver bring up, and encourages more people to write Mac OS X drivers for their devices.

Mac OS X is an evolving system, and many of these features are still in their infancy. Over time it will likely evolve into a more fine grained locking model, with certain compromises that are currently present will be phased out. The basic architecture needed to support SMP and real-time exists, and for most things the interfaces should remain stable for the foreseeable future.

References

[eC++] *eC++ Overview*,
`http://www.infoxpress.com/reviewtracker/reprints.asp?page_id=840`, (1997)

[dgux] *Digital UNIX Writing Device Drivers: Advanced Topics*,
`http://www.unix.digital.com/docs/dev_doc/ DOCUMENTATION/PDF/AA-Q7RPB.PDF`, (1996)

[iokit] *Mac OS X: I/O Kit Fundamentals*,
`http://developer.apple.com/techpubs/macosx/Darwin/IOKitFundamentals/index.html`, (2001)

[kernenv] *Mac OS X: Kernel Environment*,
`http://developer.apple.com/techpubs/macosx/Darwin/General/KernelEnvironment/ndex.html`, (2001)

[microperf] Herman Härtig, Michael Hohmuth, Jochen Liedtke, Sebastion Schoönberg, and Jean Wolter, *The Performance of μ-Kernel-Based Systems*, 16th ACM Symposium on Operating System Principles (1997)

An implementation of the Yarrow PRNG for FreeBSD

Mark R. V. Murray
FreeBSD Services, Ltd
Amersham, Buckinghamshire, UK
markm@freebsd-services.com

Abstract

Computers are by their definition predictable. The problem of obtaining good-quality random numbers is well known.

There is a great need for entropy in the running kernel, as well as in user-space. The kernel needs to randomise TCP sequences, seed keys for IPSec, randomise PIDs, and so on. Starvation of these random numbers is a critical problem. Users need random keys, random filenames, nondeterministic games, random numbers for Monte-Carlo simulation and so on.

Kelsey, Schneier and Ferguson proposed an improved algorithm for providing statistically random numbers, at the same time cryptographically protecting their sequence and state. This is the Yarrow algorithm.

This work presents an implementation of this algorithm as the entropy device (`/dev/random`) in FreeBSD's kernel.

1 Introduction

In an earlier work[Mur00], the author introduced the new entropy device to FreeBSD-CURRENT as a work-in progress. In that work, attack methodologies were briefly discussed, and the difference between the older entropy device and this device were discussed. Yarrow[KSF99] was briefly explained.

It is important to remember that this device is *not* designed to produce pure[1] random numbers. Computers do not produce enough natural randomness for that approach to be useful in entropy-consuming environments.

Instead, this device is a free-running pseudo-random number generator (PRNG), one in which great effort has been made to cryptographically protect the state of the generator. Further, the internal state is constantly perturbed with "harvested" entropy to thwart attackers.

The algorithm is divided into four parts (see Figure 1):

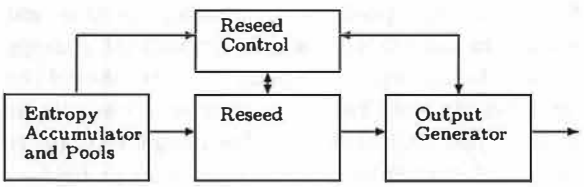

Figure 1: Simplified Yarrow Structure

- **Entropy Accumulator and Pools**
  These are used to "harvest" entropy from the running kernel. The API provided by the author is intended to be simple to use anywhere in the kernel.

- **Reseed**
  Reseeding is entirely internal to Yarrow. The author has attempted to stay as close as possible to the published algorithm.

[1] In the number-theoretic sense; the numbers remain statistically random and include environmental noise

- **Reseed Control**
  Reseeds happen in response to harvested entropy, and to reads from the entropy device. There are statistical requirements to these reseeds that are unimplemented.
- **Output Generator**
  The generator is similar to "classic" PRNG's, excepting:
  1. It uses a large, cryptographically secure hash instead of a simple feedback formula.
  2. It is perturbed on a regular basis by harvested entropy.

2 Design Issues

An API for "harvesting" entropy was needed, so that kernel programmers could easily provide such randomness their subsystem could produce. The requirements were that the API should be extensible, fast, simple and able to operate in interrupt context. Where practical, entropy sources needed the ability to be disabled at the whim of the system administrator.

256-Bit storage pools were desired, as this was deemed to hold a reasonable amount of entropy without being overly expensive. It should be remembered that Yarrow uses two accumulation "pools" (*fast* and *slow*), so this meant that up to 512 bits of environmental entropy could be held.

This decision meant that a 256-bit hashing algorithm and a 256-bit block cipher were needed. The need for a 256-bit hash ruled out using MD2, MD4, MD5 or SHA-1 unless a lengthening algorithm was also used. There were a few choices for 256-bit block ciphers, however availability (or potential availability) in the FreeBSD kernel was a limiting factor. As a suitable "natural" hash did not exist, a hash had to be constructed using block ciphers. Likely candidates were initially Blowfish and DES (reluctantly, as a block-lengthening process would be needed). Other AES candidates were considered, but as a finalist had not been selected they were not initially used.

The output generator needed to be fast, and also needed good key-setup speed, as the key is changed often. In order to preserve the strength of Yarrow, its block size was deemed to be the same size as the hash buffer. This made the choice of the encryption cipher simple, as the hash cipher could be used.

Further research[Sch96a] indicated that lengthening algorithms were most probably unwise.

2.1 Entropy Harvesting

As entropy could be found in any part of the kernel, both bottom-half and top-half, the entropy harvesting needed to be cheap, non-invasive and non-blocking.

A fixed-size circular buffer is used to accumulate entropy for later processing. If the buffer becomes full, further attempts to add entropy are ignored. The buffer is never locked when written to; this does not matter, as data corruption would be beneficial.

Entropy is added to the buffer by a subsystem calling the `random_harvest`(9) function. This is declared in sys/random.h as follows:

```
enum esource { \
    RANDOM_WRITE, RANDOM_KEYBOARD, \
    RANDOM_MOUSE, RANDOM_NET, \
    RANDOM_INTERRUPT, ENTROPYSOURCE \
};
void random_harvest(void *data, \
    u_int count, u_int bits, \
    u_int frac, enum esource source);
```

Entropy is accumulated in up to `HARVESTSIZE`[2] byte chunks.

The arguments are:

`data`	a pointer to the stochastic data
`count`	the number of bytes of data
`bits`	an estimate of the random bits
`frac`	as above, except fractional ($\frac{frac}{1024}$ bits)
`source`	the source of the entropy

[2] Currently 16

The stochastic events added to the buffer are stored in a structure:

```
struct harvest {
        u_int64_t somecounter;
        u_char entropy[HARVESTSIZE];
        u_int size, bits, frac;
        enum esource source;
};
```

The structure holds all of the information provided by `random_harvest` plus a timestamp.

The timestamp is taken from the CPU's fast counter register (like the Intel Pentium(tm) processor's TSC register). CPUs that do not have this register (like the Intel i386) use `nanotime(9)` instead. This has an unfortunate time penalty.

It is not important that this timestamp is an accurate reflection of real-world time, nor is it important that multiple CPUs in an SMP environment would have different values. It *is* important that the counter/timestamp increase quickly and linearly with time.

A count of accumulated entropy is kept, and this is used to *reseed* the output generator on occasion. The fractional entropy count supplied in the `frac` parameter is used in very low entropy situations. For example, a particular device can be said to produce 1 bit of randomness every 20 events.

Kernel programmers wishing to supply entropy from their code should extend the `enum esource` list, leaving the constant at the end of the list. Then, the randomness should be gathered and supplied as efficiently as possible.

In sys/random.h:

```
enum esource {
    RANDOM_WRITE,
    RANDOM_KEYBOARD,
    RANDOM_MOUSE,
    RANDOM_NET,
    RANDOM_INTERRUPT,
    RANDOM_MYSTUFF, /* New */
    ENTROPYSOURCE };
```

In the code to be harvested:

```
:
#include <sys/types>
:
#include <sys/random>

int
somefunc(...)
{
    :
    struct {
        u_int32_t junk;
        u_int32_t garbage;
        u_char    rubbish[8];
    } randomstuff;

    :
    randomstuff.junk = somelocaljunk;
    randomstuff.garbage = otherjunk;
    strncpy(randomstuff.rubbish, dirt, 8);
    :
    /* harvest the entropy in
     * randomstuff. Be really
     * conservative and estimate the
     * the random bit count as only 4.
     */
    random_harvest(randomstuff,
        sizeof(randomstuff), 4, 0,
        RANDOM_MYSTUFF);
    :
```

If control over the new harvesting is required, then a `sysctl` may be added to `src/sys/dev/random/randomdev.[ch]`:

```
SYSCTL_PROC(_kern_random_sys_harvest,
    OID_AUTO, interrupt,
    CTLTYPE_INT|CTLFLAG_RW,
    &harvest.mystuff, 0,
    random_check_boolean, "I",
    "Harvest mystuff entropy");
```

The call to `random_harvest` should then be made conditional on `harvest.mystuff`:

```
    :
    if (random.mystuff)
        random_harvest(randomstuff,
            sizeof(randomstuff), 4, 0,
            RANDOM_MYSTUFF);
```

Writing to the entropy device from the user's perspective (ie, writing to `/dev/random`) is similar to writing to `/dev/null`; it has no discernible effect. In actual fact, the data written is "harvested" using the harvesting calls, with the proviso that the entropy is estimated to be *nothing*. This has the effect of not causing reseeds, but perturbing the internal state anyway. If the user is the superuser, then closing the device after a write will cause an **explicit** reseed.

A kernel thread "*kthread*" constantly runs, polling the circular buffer, and if data is present, it accumulates each event alternately into the two accumulation hashes (or "entropy pools").

2.2 Accumulation Pools

An initial version of the 256-bit accumulation hash was constructed using a Davies-Meyer[Sch96b] hash with Blowfish[Sch96c] as the block cipher.

The hash works by repeatedly encrypting an initial (zero) state while cycling the hash data through the key. At each iteration, the previous value of the hash is exclusive-or-ed into the newly encrypted value.

This can be represented as:
$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$

where H_n is the n^{th} iteration of the hash result, M_j is the j^{th} fragment of the data to be hashed and $E_k(m)$ is the result of encrypting m with block cipher $E()$ and key k.

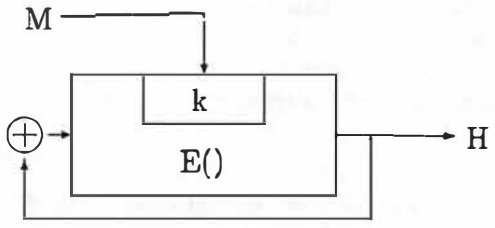

Figure 2: Davies-Meyer hash from block cipher.

While this worked, it was unbearably slow as Blowfish has an extremely expensive key schedule. Slowness was experienced as very bad kernel latency, and a kernel thread running with unacceptably high CPU usage.

The (by this time) newly released AES ("Rijndael")[NIS] algorithm was then tried, and a crude benchmark produced extremely promising results. (Here, Blowfish was replaced with Rijndael.)

The benchmark is a timed 16MB read from each device:

```
$ dd if=${DEVICE} of=/dev/null \
        count=16 bs=1048576
```

For comparison, `/dev/zero` was also read.

The time is the time in seconds for the 16MB read, and the rate is measured in KB/s.

$device	Time (s)	Rate (kB/s)
Blowfish	137.7	122
AES	6.5	2595
Zero	0.2	81861

After consulting literature [SKW+][WSB][FKL+], it was suspected that AES was the ideal algorithm, but further investigation was considered prudent, particularly as the benchmark measured output performance, not hashing performance.

The hash routines were broken out of the kernel, and various speeds were measured using alternative block ciphers. A Null algorithm and 160-bit SHA-1 were included for comparison.

The "Null" cipher simply duplicates the input data, ignoring the key:

$N_k(m) = m$

This reduced the Davies-Meyer algorithm to the XOR and data-movement parts only.

Each result represents the time taken to hash 2MB of pseudo-random data.

Algorithm	Time (s)	Rate (kB/s)
AES	3.1	461.6
Blowfish	40.2	35.2
DES	2.9	491.7
SHA-1	2.0	693.3
Null	1.8	786.7

It can be seen that AES with 256-bit keys and 256-bit blocks is approximately as fast as DES with 56-bit keys and 64-bit blocks.

160-bit SHA-1 is about 50% faster than the AES hash, but the AES hash has an approximately 50% larger capacity for storing bits.

The "Null" algorithm confirms that encryption overhead is acceptably low in comparison with other code overhead.

2.3 Output Generator

The output generator is a counter that is repeatedly encrypted, producing the output:

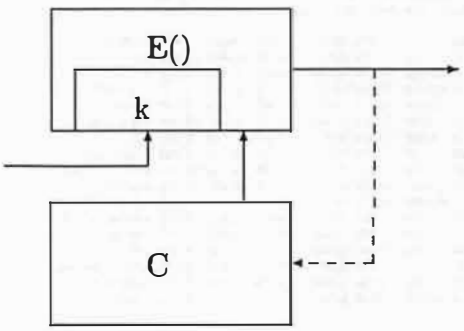

Figure 3: The Output Generator

$$O_i = E_k(C_t)$$
$$C_{t+1} = C_t + 1$$

where C_t is the (256-bit) counter[3] at time t, O_i is the i^{th} output, and $E_k(C_t)$ is the result of encrypting counter C_t using cipher $E()$ and key k.

The dashed line represents the data path during a gate event. The key "k" is inserted during a *reseed*. This is the point at which environmental noise ("harvested" entropy) is used.

To compromise the output generator, a key compromise of the cipher is necessary. This is computationally difficult; nevertheless, to thwart this, the counter is regularly replaced with data from the output stream:

$$C_{t+1} = C_t + 1$$
$$C_{t+1} = E_k(C)$$

The data thus used is *not* used as part of the output. This is called a *gate event*, and it happens at a time configurable by the system administrator via `sysctl(9)`. It defaults to happening every 10 blocks. If a user process reads less than a 256-bit block, the remainder is cached for future reads.

To show that the output was statistically acceptable, some tests were done.

A simple histogram of 8M single-byte values was plotted:

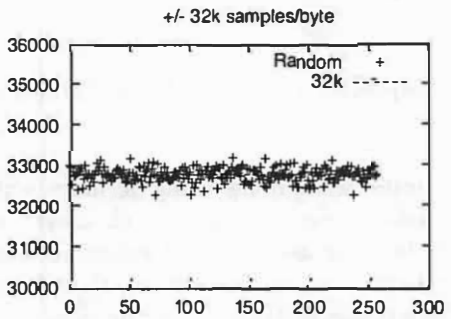

Figure 4: Spectrum of 8M 8-bit values

A straight line was fitted to this data, and was found to substantiate the fact that the slope was ≈ 0 and the mean value was $\approx 32k$.

[3]Internal to the FreeBSD kernel, the 256-bit value is represented as a structure containing four 64-bit unsigned integers. Only 64 bits are incremented. The author does not believe this is a problem.

The spread of values around $32k$ was plotted, and the distribution found to be reassuringly normal:

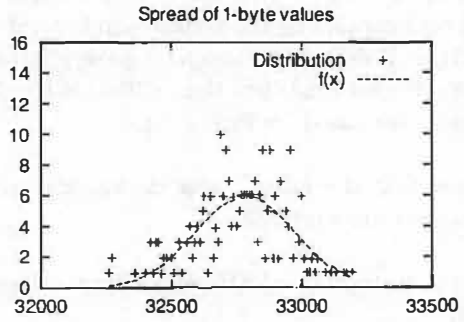

Figure 5: Distribution of values around expected norm of 32k

This corresponded to a mean (μ) of 32759.5 and a σ of 187.2

Further tests were done using a more sophisticated random number "torture chamber" called Diehard[Mar]. Its use produced voluminous output which indicated, on careful perusal, that the generator's output was statistically acceptable.

It must be noted that the output generator *does not block*. This is intentional.

2.4 Reseed Control

This is the trickiest part of the algorithm to write. The Yarrow specification mandates three separate estimates of incoming entropy "harvest-units":

1. A programmer-supplied estimate. This has been *very* conservatively set. This is given as a constant to each entropy-harvesting call.

2. A system-wide "density". This is set at $\frac{1}{2}$, meaning no sample of N bits can supply more than $\frac{N}{2}$ bits of entropy.

3. A statistically determined, per-source continuous estimate. This is unimplemented, as the mechanism for doing the statistical estimation has been deemed too expensive for the kernel.

The algorithm states that the *lowest* of these three is taken as the entropy supplied for the individual unit. The author has endeavoured to ensure that the programmer-supplied estimate will always be low enough.

3 Impact on the Running System

The running device has great potential to be very invasive to the running kernel, as early experiments with slow ciphers showed. In the current code, however, the system is proving to be no such hindrance.

```
last pid: 19524;  load averages:  0.26,  0.22,  0.18    up 3+09:01:43  21:52:53
92 processes:  3 running, 74 sleeping, 15 waiting
CPU states:  4.3% user,  0.0% nice,  2.3% system,  0.4% interrupt, 93.0% idle
Mem: 27M Active, 5538K Inact, 18M Wired, 4348K Cache, 14M Buf, 4856K Free
Swap: 68M Total, 34M Used, 33M Free, 50% Inuse

  PID USERNAME PRI NICE   SIZE    RES STATE    TIME   WCPU    CPU COMMAND
   10 root     -16    0     0K    12K RUN     59.9H 86.47% 88.47% idle
18128 root      96    0 16636K  5932K select  16:37  1.61%  1.61% XFree86
18169 mark      96    0 15664K  3216K select   4:53  1.61%  1.61% kdeinit
19217 mark      96    0 17116K  4540K select   1:47  0.39%  0.39% kdeinit
18227 mark      96    0 10872K  5916K select   1:59  0.29%  0.29% xemacs-21.1.1
19524 mark      96    0  2096K  1144K RUN      0:00  0.76%  0.20% top
   22 root     -64 -183     0K    12K WAIT    40:57  0.10%  0.10% irq14: ata0
   12 root     -48 -187     0K    12K RUN     20:06  0.10%  0.10% swi6: tty:sio
18205 mark      96    0 17836K  5928K select   1:38  0.10%  0.10% kdeinit
18203 mark      96    0 21428K  4080K select   1:07  0.10%  0.10% kdeinit
    6 root      20    0     0K    12K syncer   4:02  0.00%  0.00% syncer
   14 root      76    0     0K    12K sleep    3:19  0.00%  0.00% random
18183 mark      60  -36  5244K  2044K select   3:09  0.00%  0.00% artsd
   15 root     -28 -147     0K    12K WAIT     2:58  0.00%  0.00% swi5: task qu
18810 mark      96    0  9728K  3076K select   1:30  0.00%  0.00% acroread
18208 mark      96    0 16372K  3964K select   1:21  0.00%  0.00% kdeinit
```

Figure 6: Snapshot of a running system

This snapshot of a running FreeBSD workstation shows that the **random** process (the *kthread* that runs the reseed process) has approximately the same impact on the system as the **syncer** process, ie negligible.

The use of random numbers by security-conscious engineers has been taken into account over and above the concerns of the professional cryptographic community. Speed was deemed to be more important than the production of number-theoretic-quality random numbers (eg: suited to generating one-time-pads). It is believed that FreeBSD is used by many more system-administrators than professional cryptographers.

The author is, however, appreciative of the concerns of those who would want a more austere presentation of random numbers from the operating system.

Those members of the community are considered to be a specialist minority, though.

4 Future plans

There are two main expansion areas in the FreeBSD entropy device.

1. More entropy harvesting. Any "cheap" entropy that may be found in the kernel may be used in the future. The user community is encouraged to submit likely sources. The author has provisional code to harvest entropy from Intel chipsets fitted with hardware random number generators.
2. Provision of a "distilled" device for those who wish to be assured of an "entropy-in = entropy-out" conservation-of-entropy device. This needs to be conservative enough to not provide a denial-of-service attack by its very existence.

5 Thanks

Thanks are due to Sue Bourne and Brian Somers for proofreading and helpful comments.

Thanks are also due to FreeBSD Services, Ltd for giving me the time to produce this work.

My fondest thanks are also given to my father. Thanks, Dad. I'll miss you.

References

[FKL+] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Mike Stay, David Wagner, and Doug Whiting. Improved cryptanalysis of rijndael. http://www.counterpane.com.

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. Sixth Annual Workshop on Selected Areas in Cryptography, August 1999.

[Mar] George Marsaglia. Diehard. http://www/stat.fsu.edu/~geo/diehard.html.

[Mur00] Mark R. V. Murray. Effective entropy from the freebsd kernel. In *BSDCon*, pages 92–98, 2000.

[NIS] NIST. The aes algorithm (rijndael) information. http://csrc.nist.gov/encryption/aes/rijndael/.

[Sch96a] Bruce Schneier. *Applied Cryptography*, pages 430–431. Wiley, second edition, 1996.

[Sch96b] Bruce Schneier. *Applied Cryptography*, pages 446–455. Wiley, second edition, 1996.

[Sch96c] Bruce Schneier. *Applied Cryptography*, pages 336–339. Wiley, second edition, 1996.

[SKW+] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. Performance comparison of the aes submissions. http://www.counterpane.com.

[WSB] Doug Whiting, Bruce Schneier, and Steve Bellovin. Aes key agility issues in high-speed ipsec implementations. http://www.counterpane.com.

Running "Fsck" in the Background

Marshall Kirk McKusick
Author and Consultant

Abstract

Traditionally, recovery of a BSD fast filesystem after an uncontrolled system crash such as a power failure or a system panic required the use of the filesystem checking program, "fsck". Because the filesystem cannot be used while it is being checked by "fsck", a large server may experience unacceptably long periods of unavailability after a crash.

Rather than write a new version of "fsck" that can run on an active filesystem, I have added the ability to take a snapshot of a filesystem partition to create a quiescent filesystem on which a slightly modified version of the traditional "fsck" can run.

A key feature of these snapshots is that they usually require filesystem write activity to be suspended for less than one second. The suspension time is independent of the size of the filesystem. To reduce the number and types of corruption, *soft updates* were added to ensure that the only filesystem inconsistencies are lost resources. With these two additions it is now possible to bring the system up immediately after a crash and then run checks to reclaim the lost resources on the active filesystems.

Background "fsck" runs by taking a snapshot and then running its traditional first four passes to calculate the correct bitmaps for the allocations in the filesystem snapshot. From these bitmaps, "fsck" finds any lost resources and invokes special system calls to reclaim them in the underlying active filesystem.

1. Background and Introduction

Traditionally, recovery of the BSD fast filesystem [McKusick, et al., 1996] after an uncontrolled system crash such as a power failure or a system panic required the use of the "fsck" filesystem checking program [McKusick & Kowalski, 1994]. "Fsck" would inspect all the filesystem metadata (bitmaps, inodes, and directories) and correct any inconsistencies that were found. As the metadata comprises about three to five percent of the disk space, checking a large filesystem can take an hour or longer. Because the filesystem is inaccessible while it is being checked by "fsck", a large server may experience unacceptably long periods of unavailability after a crash.

Many methods exist for solving the metadata consistency and recovery problem [Ganger et al, 2000; Seltzer et al, 2000]. The solution selected for the fast filesystem is *soft updates*. Soft updates control the ordering of filesystem updates such that the only inconsistencies in the on-disk representation are that free blocks and inodes may be claimed as "in use" in the on-disk bitmaps when they are really unused.

Soft updates eliminates the need to run "fsck" after a system crash except in the rare event of a hardware failure in the disk on which the filesystem resides. Since the only filesystem inconsistency is lost blocks and inodes, the only ill effect from running on the filesystem after a crash is that part of the filesystem space that should have been available will be lost. Any time the lost space from crashes reaches an unacceptable level, the filesystem must be taken offline long enough to run "fsck" to reclaim the lost resources.

Because many server systems need to be available 24x7, there is never an available multi-hour time when a traditional version of "fsck" can be run. Thus, I have been motivated to develop a filesystem check program that can run on an active filesystem to reclaim the lost resources.

The first approach that I considered was to write a new utility that would operate on an active filesystem to identify the lost resources and reclaim them. Such a utility would fall into the class of real-time garbage collection. Despite much research and

literature, garbage collection of actively used resources remains challenging to implement. In addition to the complexity of real-time garbage collection, the new utility would need to recreate much of the functionality of the existing "fsck" utility. Having a second utility would lead to additional maintenance complexity as bugs or feature additions would need to be implemented in two utilities rather than just one.

To avoid the complexity and maintenance problems, I decided to take an approach that would enable me to use most of the existing "fsck" program.

"Fsck" runs on the assumption that the filesystem it is checking is quiescent. To create an apparently quiescent filesystem, I added the ability to take a *snapshot* of a filesystem partition [McKusick & Ganger, 1999]. Using copy-on-write, a snapshot provides a filesystem image frozen at a specific point in time. The next section describes the details on how snapshots are taken and managed.

As lost resources will not be used until they have been found and marked as free in the bitmaps, there are no limits on how long a background reclamation scheme can take to find and recover them. Thus, I can run the traditional "fsck" over a snapshot to find all the missing resources even if it takes several hours and the filesystem is being actively changed.

Snapshots may seem a more complex solution to the problem of running space reclamation on an active filesystem than a more straight forward garbage-collection utility. However, their cost (about 1300 lines of code) is amortized over the other useful functionality: the ability to do reliable dumps of active filesystems and the ability to provide back-ups of the filesystem at several times during the day. This functionality was first popularized by Network Appliance [Hitz et al, 1994; Hutchinson et al, 1999].

Snapshots enable the safe backup of live filesystems. When **dump** notices that it is being asked to dump a mounted filesystem, it can simply take a snapshot of the filesystem and run over the snapshot instead of on the live filesystem. When **dump** completes, it releases the snapshot.

Periodic snapshots can be made accessible to users. When taken every few hours during the day, they allow users to retrieve a file that they wrote several hours earlier and later deleted or overwrote by mistake. Snapshots are much more convenient to use than dump tapes and can be created much more frequently.

To make a snapshot accessible to users through a traditional filesystem interface, BSD uses the memory-disk driver, *md*. The **mdconfig** command takes a snapshot file as input and produces a **/dev/md0** character-device interface to access it. The **/dev/md0** character device can then be used as the input device for a standard BSD FFS mount command, allowing the snapshot to appear as a replica of the frozen filesystem at whatever location in the namespace that the system administrator chooses to mount it. Thus, the following script could be run at noon to create a mid-day backup of the **/usr** filesystem and make it available at **/backups/usr/noon**:

```
# Take the snapshot
mount -u -o snapshot /usr/snap.noon /usr
# Attach it to /dev/md0
mdconfig -a -t vnode -u 0 -f /usr/snap.noon
# Mount it for user access
mount -r /dev/md0 /backups/usr/noon
```

When no longer needed, it can be removed with:

```
# Unmount snapshot
umount /backups/usr/noon
# Detach it from /dev/md0
mdconfig -d -u 0
# Delete the snapshot
rm -f /usr/snap.noon
```

2. Creating a Filesystem Snapshot

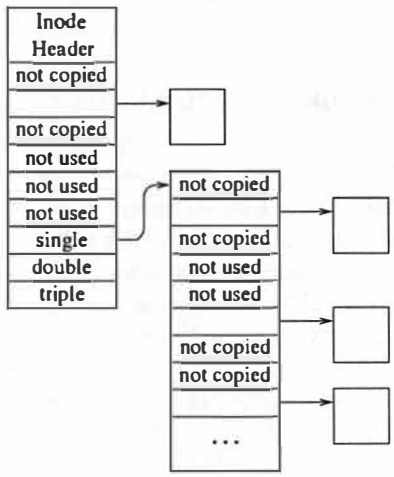

Figure 1: *Structure of a snapshot file*

A filesystem *snapshot* is a frozen image of a filesystem at a given instant in time. Implementing snapshots in the BSD fast filesystem has proven to be straightforward. Taking a snapshot entails the following steps:

I) A *snapshot file* is created to track later changes to the filesystem; a snapshot file is shown in Fig. 1. This snapshot file is initialized to the size of the

filesystem's partition, and its file block pointers are marked as zero which means "not copied." A few strategic blocks are allocated, such as those holding copies of the superblock and cylinder group maps.

2) A preliminary pass is made over each of the cylinder groups to copy it to its preallocated backing block. Additionally, the block bitmap in each cylinder group is scanned to determine which blocks are free. For each free block that is found, the corresponding location in the snapshot file is marked with a distinguished block number (1) to show that the block is "not used." There is no need to copy those unused blocks if they are later allocated and written.

3) The filesystem is marked as "wanting to suspend." In this state, processes that wish to invoke system calls that will modify the filesystem are blocked from running, while processes that are already in progress on such system calls are permitted to finish them. These actions are enforced by inserting a gate at the top of every system call that can write to a filesystem. The set of gated system calls includes "write", "open" (when creating or truncating), "fhopen" (when creating or truncating), "mknod", "mkfifo", "link", "symlink", "unlink", "chflags", "fchflags", "chmod", "lchmod", "fchmod", "chown", "lchown", "fchown", "utimes", "lutimes", "futimes", "truncate", "ftruncate", "rename", "mkdir", "rmdir", "fsync", "sync", "unmount", "undelete", "quotactl", "revoke", and "extattrctl". In addition gates must be added to "pageout", "ktrace", local domain socket creation, and core dump creation. The gate tracks activity within a system call for each mounted filesystem. A gate has two purposes. The first is to suspend processes that want to enter the gated system call during periods that the filesystem that the process wants to modify is suspended. The second is to keep track of the number of processes that are running inside the gated system call for each mounted filesystem. When a process enters a gated system call, a counter in the mount structure for the filesystem that it wants to modify is incremented. When the process exits a gated system call, the counter is decremented.

4) The filesystem's status is changed from "wanting to suspend" to "fully suspended." This status change is done by allowing all system calls currently writing to the filesystem being suspended to finish. The transition to "fully suspended" is complete when the count of processes within gated system calls drops to zero.

5) The filesystem is synchronized to disk as if it were about to bc unmounted.

6) Any cylinder groups that were modified after they were copied in step two are recopied to their preallocated backing block. Additionally, the block bitmap in each recopied cylinder group is rescanned to determine which blocks were changed. Newly allocated blocks are marked as "not copied" and newly freed blocks are marked as "not used." The details on how these modified cylinder groups are identified is described below. The amount of space initially claimed by a snapshot is small, usually less than a tenth of one percent. Actual snapshot file space utilization is given in section 4.

7) With the snapshot file in place, activity on the filesystem resumes. Any processes that were blocked at a gate are awakened and allowed to proceed with their system call.

8) Blocks that had been claimed by any snapshots that existed at the time that the current snapshot was taken are expunged from the new snapshot for reasons described below.

During steps three through six, all write activity on the filesystem is suspended. Steps three and four complctc in at most a few milliseconds. The time for step five is a function of the number of dirty pages in the kernel. It is bounded by the amount of memory that is dedicated to storing file pages. It is typically less than a second and is independent of the size of the filesystem. Typically step six needs to recopy only a few cylinder groups, so it also completes in less than a second.

The splitting of the bitmap copies between steps two and six is the way that I avoid having the suspend time be a function of the size of the filesystem. By making our primary copy pass while the filesystem is still active, and then having only a few cylinder groups in need of recopying after it has been suspended, I keep the suspend time down to a small and usually filesystem size independent time.

The details of the two-pass algorithm are as follows. Before starting the copy and scan of all the cylinder groups, the snapshot code allocates a "progress" bitmap whose size is equal to the number of cylinder groups in the filesystem. The purpose of the "progress" bitmap is to keep track of which cylinder groups have been scanned. Initially, all the bits in the "progress" map are cleared. The first pass is completed in step two before the filesystem is suspended. In this first pass, all the cylinder groups are scanned. When the clinder group is read, its

corresponding bit is set in the "progress" bitmap. The cylinder group is then copied and its block map is consulted to update the snapshot file as described in step two. Since the filesystem is still active, filesystem blocks may be allocated and freed while the cylinder groups are being scanned. Each time a cylinder group is updated because of a block being allocated or freed, its corresponding bit in the "progress" bitmap is cleared. Once this first pass over the cylinder groups is completed, the filesystem is "suspended."

Step six now becomes the second pass of the algorithm. The second pass need only identify and update the snapshot for any cylinder groups that were modified after it handled them in the first pass. The changed cylinder groups are identified by scanning the "progress" bitmap and rescanning any cylinder groups whose bits are zero. Although every bitmap would have to be reprocessed in the worst case, in practice only a few bitmaps need to be recopied and checked.

3. Maintaining a Filesystem Snapshot

Each time an existing block in the filesystem is modified, the filesystem checks whether that block was in use at the time that the snapshot was taken (i.e., it is not marked "not used"). If so, and if it has not already been copied (i.e., it is still marked "not copied"), a new block is allocated from among the "not used" blocks and placed in the snapshot file to replace the "not copied" entry. The previous contents of the block are copied to the newly allocated snapshot file block, and the modification to the original is then allowed to proceed. Whenever a file is removed, the snapshot code inspects each of the blocks being freed and claims any that were in use at the time of the snapshot. Those blocks marked "not used" are returned to the free list.

When a snapshot file is read, reads of blocks marked "not copied" return the contents of the corresponding block in the filesystem. Reads of blocks that have been copied return the contents in the copied block (e.g., the contents that were stored at that location in the filesystem at the time that the snapshot was taken). Writes to snapshot files are not permitted. When a snapshot file is no longer needed, it can be removed in the same way as any other file; its blocks are simply returned to the free list and its inode is zeroed and returned to the free inode list.

Snapshots may live across reboots. When a snapshot file is created, the inode number of the snapshot file is recorded in the superblock. When a filesystem is mounted, the snapshot list is traversed and all the listed snapshots are activated. The only limit on the number of snapshots that may exist in a filesystem is the size of the array in the superblock that holds the list of snapshots. Currently, this array can hold up to twenty snapshots.

Multiple snapshot files can exist concurrently. As described above, earlier snapshot files would appear in later snapshots. If an earlier snapshot is removed, a later snapshot would claim its blocks rather than allowing them to be returned to the free list. This semantic means that it would be impossible to free any space on the filesystem except by removing the newest snapshot. To avoid this problem, the snapshot code goes through and expunges all earlier snapshots by changing its view of them to being zero length files. With this technique, the freeing of an earlier snapshot releases the space held by that snapshot.

When a block is overwritten, all snapshots are given an opportunity to copy the block. A copy of the block is made for each snapshot in which the block resides. Overwrites typically occur only for inode and directory blocks. File data is usually not overwritten. Instead, a file will be truncated and then reallocated as it is rewritten. Thus, the slow and I/O intensive block copying is infrequent.

Deleted blocks are handled differently. The list of snapshots is consulted. When a snapshot is found in which the block is active ("not copied"), the deleted block is claimed by that snapshot. The traversal of the snapshot list is then terminated. Other snapshots for which the block are active are left with an entry of "not copied" for that block. The result is that when they access that location, they will still reference the deleted block. Since snapshots may not be modified, the block will not change. Since the block is claimed by a snapshot, it will not be allocated to another use. If the snapshot claiming the deleted block is deleted, the remaining snapshots will be given the opportunity to claim the block. Only when none of the remaining snapshots wants to claim the block (i.e., it is marked "not used" in all of them) will it be returned to the freelist.

4. Snapshot Performance

The experiments described in this section and the background "fsck" performance section used the following hardware/software configuration:

Computer: Dual Processor using two Celeron 350MHz CPUs. The machine has 256Mb of main memory.

O/S: FreeBSD 5.0-current as of December 30, 2001

I/O Controller: Adaptec 2940 Ultra2 SCSI adapter

Disk: Two <IBM DDRS-39130D DC1B> Fixed Direct Access SCSI-2 device, 80MB/s transfers, Tagged Queuing Enabled, 8715MB, 17,850,000 512 byte sectors: 255H 63S/T 1111C

Small Filesystem: 0.5Gb, 8K block, 1K fragment, 90% full, 70874 files, initial snapshot size 0.392Mb (0.08% of filesystem space).

Large Filesystem: 7.7Gb, 16K block, 2K fragment, 90% full, 520715 files, initial snapshot size 2.672Mb (0.03% of filesystem space).

Load: Four continuously running simultaneous Andrew benchmarks that create a moderate amount of filesystem activity intermixed with periods of CPU intensive activity [Howard et al, 1988].

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	0.7 sec	0.1 sec	0.025 sec
7.7Gb	3.5 sec	0.4 sec	0.034 sec

Table 1: *Snapshot times on an idle filesystem*

Table 1 shows the time to take a snapshot on an idle filesystem. The elapsed time to take a snapshot is proportional to the size of the filesystem being snapshotted. However, nearly all the time to take a snapshot is spent in steps one, two, and eight. Because the filesystem permits other processes to modify the filesystem during steps one, two, and eight, this part of taking a snapshot does not interfere with normal system operation. The "suspend time" column shows the amount of real-time that processes are blocked from executing system calls that modify the filesystem. As Table 1 shows, the period during which write activity is suspended, and thus apparent to processes in the system, is short and does not increase proportionally to filesystem size.

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	3.7 sec	0.1 sec	0.027 sec
7.7Gb	12.1 sec	0.4 sec	0.036 sec

Table 2: *Snapshot times on an active filesystem*

Table 2 shows the times to snapshot a filesystem that has four active concurrent processes running. The elapsed time rises because the process taking the snapshot has to compete with the other processes for access to the filesystem. Note that the suspend time has risen slightly, but is still insignificant and does not increase in proportion to the size of the filesystem under test. Instead, it is a function of the level of write activity present on the filesystem.

Filesystem Size	Elapsed Time	CPU Time
0.5Gb	0.5 sec	0.02 sec
7.7Gb	2.3 sec	0.09 sec

Table 3: *Snapshot removal time on an idle filesystem*

Table 3 shows the times to remove a snapshot on an idle filesystem. The elapsed time to remove a snapshot is proportional to the size of the filesystem being snapshotted. The filesystem does not need to be suspended to remove a snapshot.

Filesystem Size	Elapsed Time	CPU Time
0.5Gb	1.4 sec	0.03 sec
7.7Gb	4.9 sec	0.10 sec

Table 4: *Snapshot removal time on an active filesystem*

Table 4 shows the times to remove a snapshot on a filesystem that has four active concurrent processes running. The elapsed time rises because the process removing the snapshot has to compete with the other processes for access to the filesystem. The filesystem does not need to be suspended to remove a snapshot.

5. Implementation of Background "Fsck"

Background "fsck" runs by taking a snapshot then running its traditional algorithms over the snapshot. Because the snapshot is taken of a completely quiescent filesystem, all of whose dirty blocks have been written to disk, the snapshot appears to "fsck" to be exactly like an unmounted raw disk partition. "Fsck" runs in five passes that can be summarized as follows:

1) Scan all the allocated inodes to accumulate a list of all their allocated blocks. The fast filesystem preallocates all the inodes that the filesystem will ever be able to use at the time that the filesystem is created. The superblock contains information on where all the preallocated inodes can be found. Thus, "fsck" can find all of them without need of any additional information such as an index-of-inodes file or any of the filesystem directory structure. As part of scanning all the allocated inodes, "fsck" also identifies all the inodes that are allocated as directories for use in the next three passes. When the first pass completes, "fsck" has a list of all the allocated blocks and inodes.

2) Scan all the directories found in pass one. For each entry found in a directory, increment the reference count in the inode that it references. If the referenced inode is a directory, verify that its dot-dot entry points back properly. Also note that an entry to the directory has been found. The one exception to the dot-dot rule is for the root of the filesystem (inode two) whose dot-dot entry should point to itself.

3) Check that every directory (except the root) was found during the second pass. If any directories were not found, they have been somehow lost from the main tree. In traditional "fsck" any lost directories would be placed into the **lost+found** directory. When running with soft updates, unreferenced directories only occur if they were in the process of being deleted. So, if any turn up in pass three, "fsck" marks them for deletion in pass four.

4) If any inodes have a higher reference count than the number of directory entries that reference them, their reference count is adjusted to reflect the correct number of references. Such errors occur when a directory entry has been deleted but the system crashed before updating the on-disk reference count in the inode. The reference count should never need to be increased; if such a condition is found, "fsck" marks the filesystem as needing manual intervention and exits. There is a special case in which no entries for an inode were found. When running with soft updates, an unreferenced inode can only happen if the file was in the process of being deleted. Thus, background "fsck" requests the kernel to decrement the reference count on the inode to zero. The kernel then follows the usual code path for inodes with a zero reference count that releases the inode's claimed blocks and then releases the inode itself. The freed inode and any blocks that it claimed are removed from the list of valid inodes and blocks found in the first pass. In traditional "fsck" running on a filesystem that is not using soft updates, unreferenced inodes are placed in the **lost+found** directory.

5) The list of valid inodes and blocks determined in the first pass and updated in the fourth pass is compared against the bitmaps in the cylinder group maps. If there are any disagreements, the bitmaps in the cylinder groups are updated with the correct entries.

The new background version of "fsck" cannot update the on-disk image of the filesystem as the filesystem state will be different from that of the snapshot. Additionally, a user-level program cannot obtain the kernel-level locks needed to provide consistent updates of filesystem data structures. To ensure that the filesystem state is set using the appropriate locking protocol, a set of system calls was added to enable "fsck" to pass the resource-update requests to the kernel so that they can be made under the appropriate lock.

Five operations were implemented in the kernel:

1) set/clear superblock flags
2) adjust an inode block count
3) adjust an inode reference count
4) free a range of inodes
5) free a range of blocks/fragments

The background version of "fsck" is derived from the traditional disk-based "fsck" by augmenting the traditional "fsck" with calls to the kernel functions in place of writes to the filesystem partition. If at any time, "fsck" finds any inconsistencies other than lost blocks and inodes or high block or reference counts, then either a hardware or software error has occurred and a traditional execution of "fsck" needs to be run. "Fsck" sets a superblock flag (using operation 1) to force this check to be done before the next time that the filesystem is mounted. Optionally, it can forcibly downgrade a corrupted filesystem to read-only. In more detail, the changes to each pass of "fsck" are as follows:

1) After scanning each allocated inode, "fsck" compares the number of blocks that it claims with its count of the number of blocks that it is using. If these values differ, the traditional "fsck" would write back the inode with the updated value. Incorrect block counts typically occur when a partially truncated inode is encountered. The background "fsck" uses operation 2 to have the kernel adjust the count. As the file may be actively growing, the adjustment is done as an increment or decrement to the current value rather than setting an absolute value. No other changes to pass one are required.

2) No changes to pass two are required.

3) If any orphaned directories are found in pass three, they are assumed to have been in the process of being deleted. Thus they are marked for deletion in pass four.

4) In the traditional "fsck", inodes with high but non-zero reference counts need to have their reference counts adjusted. Inodes with zero reference

counts need to be zeroed out on disk. With the background "fsck" these two operations can be subsumed into a single system call (operation 3) that adjusts the reference count on an inode. If the count is reduced to zero, the kernel will deallocate and zero out the inode as part of its normal course of operation. So, no additional work is required of "fsck". As with the adjustment of the block count in pass one, the reference count on the inode may have changed since the snapshot because of ongoing filesystem activity. Thus, the adjustment is given as a delta rather than as an absolute value to ensure that the inode retains the correct reference count.

5) The final pass taken by the traditional "fsck" is to rewrite the filesystem bitmaps to reflect the allocations that it has found. As an active filesystem will have continued to allocate and free resources, the state of the bitmaps calculated in the snapshot by the background "fsck" will not be correct, so it cannot write them back in their entirety. However, it can figure out which blocks and inodes are lost by doing an exclusive-or of the bitmaps in the snapshot with the bitmaps that it has calculated. The resulting non-zero bits will be the lost resources. Having determined which resources are lost, "fsck" must cause the live bitmaps to be repaired.

   If an inode has been zeroed on the disk, but has not been marked free in the bitmaps, then it is so marked (using operation 4). If there were any unclaimed blocks that were not released when adjusting the inode reference counts, they are freed (using operation 5). These unclaimed blocks arise from an inode that was zeroed on disk, but whose formerly claimed blocks were not freed before the system crashed.

The final step after a successful background "fsck" run is to update the filesystem status in the superblock. There are two flags in the superblock that track the state of a filesystem. The first is the "clean" flag that is set when a filesystem is unmounted (by the system administrator or at system shutdown) and cleared while it is mounted with writing enabled. The second is the "unclean-at-mount" flag that is described below.

The "clean" flag is used by the traditional "fsck" to decide which filesystems need to be checked. Those filesystems with the flag set are skipped; those filesystems with the flag clear are checked. Following a successful check, the "clean" flag is set. Before soft updates, the kernel did not allow unclean filesystems (e.g., filesystems with the "clean" flag cleared) to be mounted for writing as the corruption could cause the system to panic.

The "unclean-at-mount" flag was added as part of soft updates. Unclean filesystems running with soft updates are safe to mount with writing permitted. However, the system needs to remember that some cleanup may be required. Thus, the "unclean-at-mount" flag gets set when an unclean filesystem is mounted (e.g., mounted with writing enabled without the "clean" flag having been set). The "unclean-at-mount" flag serves two purposes. First, when a filesystem with the "unclean-at-mount" is unmounted, the "clean" flag is not set to show that cleaning is still required. Second it tracks the filesystems that need cleaning. By the time that background "fsck" is run, all the filesystems are mounted so none will have their "clean" flag set. Thus, the "unclean-at-mount" flag is used by the background "fsck" to distinguish which filesystems need to be checked. Those filesystems with the flag set are checked; those filesystems with the flag clear are skipped.

So, the final step after a successful run of a background "fsck" is to clear the "unclean-at-mount" bit in the superblock (using operation 1) so that the filesystem will be marked "clean" when it is unmounted by the system administrator or at system shutdown.

6. Operation of Background "Fsck"

Traditionally, "fsck" is invoked before the filesystems are mounted and all checks are synchronously done to completion at that time. If background checking is available, "fsck" is invoked twice. It is first invoked at the traditional time, before the filesystems are mounted, with the –F flag to do checking on all the filesystems that cannot do background checking. Filesystems that require traditional checking are those that are not running with soft updates and those that will not be mounted at system startup (e.g., those marked "noauto" in **/etc/fstab**). It is then invoked a second time, after the system has completed going multiuser, with the –B flag to do checking on all the filesystems that can do background checking. Unlike the foreground checking, the background checking is started asynchronously so that other system activity can proceed even on the filesystems that are being checked.

The "fsck" program is really just a front end that reads the **/etc/fstab** file and determines which filesystems need to be checked. For each filesystem to be checked, the appropriate back end is invoked.

For the fast filesystem, "fsck_ffs" is invoked. If "fsck" is invoked with neither the –F nor the –B flag, it runs in traditional mode and checks every listed filesystem. Otherwise it is invoked with the –F to request that it run in foreground mode. In foreground mode, the check program for each filesystem is invoked with the –F flag to determine whether it wishes to run as part of the boot up sequence, or if it is able to do its job in background after the system is up and running. A non-zero exit code indicates that it wants to run in foreground and it is invoked again with neither the –F nor the –B flag so that it will run in its traditional mode. A zero exit code indicates that it is able to run later in background and just a deferred message is printed. The "fsck" program attempts to run as many checks in parallel as possible. Typically it can run a check on each disk in parallel.

After the system has gone multiuser, "fsck" is invoked with the –B flag to request that it run in background mode. The check program for each filesystem is invoked with the –F flag to determine whether it wishes to run as part of the boot up sequence, or if it is able to do its job in background after the system is up and running. A non-zero exit code indicates that it wanted to run in foreground that is assumed to have been done, so the filesystem is skipped. A zero exit code indicates that it is able to run in background so the check program is invoked with the –B flag to indicate that a check on the active filesystem should be done. When running in background mode, only one filesystem at a time will be checked. To further reduce the load on the system, the background check is typically run at a *nice* value of plus four.

The "fsck_ffs" program does the actual checking of the fast filesystem. When invoked with the –F flag, "fsck_ffs" determines whether the filesystem needs to be checked immediately in foreground or if its checking can be deferred to background. To be eligible for background checking it must have been running with soft updates, not have been marked as needing a foreground check, and be mounted and writable when the background check is to be done. If these conditions are met, then "fsck_ffs" exits with a zero exit status. Otherwise it exits with a non-zero exit status. If the filesystem is clean, it will exit with a non-zero exit status so that the clean status of the filesystem can be verified and reported during the foreground checks. Note that, when invoked with the –F flag, no checking is done. The only thing that "fsck_ffs" does is to determine whether a foreground or background check is needed and exit with an appropriate status code.

When "fsck_ffs" is invoked with the –B flag, a check is done on the specified and possibly active filesystem. The potential set of corrections is limited to those available when running in preen mode (as further detailed in the previous section). If unexpected errors are found, the filesystem is marked as needing a foreground check and "fsck_ffs" exits without attempting any further checking.

7. Background "Fsck" Performance

Over many years of tuning and refinement, "fsck" has been optimized to minimize and cluster its I/O requests. Further, its data structures have been tuned to the point where it consumes little CPU time and thus its running time is totally dominated by the time that it takes to do the needed I/O. The performance of background "fsck" is almost identical to that of the traditional "fsck". Since it is using the same algorithms, the number and pattern of its I/O requests are identical to the traditional program. Though the update requests are done through kernel calls rather than direct writes to the disk, the kernel calls typically execute the same (or through the benefits of soft updates) slightly fewer disk writes. The main added delay of background "fsck" is the time required to take and remove a snapshot of the filesystem. The time for the snapshot operation has already been covered in section 4 above.

Because of "fsck"'s I/O clustering, it is capable of using nearly all the bandwidth of a disk. Although background "fsck" only checks one filesystem partition at a time (as compared to traditional "fsck" that checks all separate disks containing filesystems in parallel), even a single instance of "fsck" can cause seriously increased latency to processes trying to access files on the filesystem (or anything else on the same disk) that is being checked.

As there is no urgency in completing the space reclamation, background "fsck" is usually run at lower priority than other processes. The usual way to reduce priority is to *nice* the process to some positive value which results in it getting a lower priority for the CPU. Because "fsck" is nearly completely I/O bound, giving it a lower CPU priority has almost no effect on the time in which it runs and hence in its rate of issuing I/O requests.

As a general solution to reducing the resource usage of I/O bound processes such as background "fsck", a small change has been made to the disk strategy routine. When an I/O request is posted, the disk strategy routine first checks whether the process is running at a positive *nice*. If it is, and there are any

other outstanding I/O requests for the disk, the process is put to sleep for *nice* hundredths of a second. Thus, a process running at a *nice* of four will sleep for forty millisecond each time it makes a disk I/O request. Such a process will be able to do at most twenty-five disk I/O requests per second – about a third of the bandwidth of a current technology disk. At the maximum *nice* value of twenty, a process is limited to five I/O requests per second which is low enough to be almost unnoticeable by other processes competing for access to the disk. Because the slow-down is imposed only when there are other outstanding disk requests, I/O bound processes can run at full speed on an otherwise idle system.

Filesystem Size	Elapsed Time	CPU Time
0.5Gb	5.8 sec	1.1 sec
7.7Gb	50.9 sec	8.3 sec

Table 5: *Traditional fsck times on an idle filesystem*

Table 5 shows the times to run the traditional "fsck" on a filesystem that is otherwise idle. It is running with a *nice* value of zero. It is the only process active on the system, so represents a lower bound on the time that a traditional disk check can be done.

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	8.1 sec	2.5 sec	0.025 sec
7.7Gb	61.5 sec	14.9 sec	0.050 sec

Table 6: *Background fsck times on an idle filesystem*

Table 6 shows the times to run background "fsck" on a filesystem that is otherwise idle. It is running with a *nice* value of zero. It is the only process active on the system, so represents a lower bound on the time that a background disk check can be done. Note that its running time is only slightly greater than would be expected by adding the time to take and remove a snapshot (see Table I and Table 3) to the running time of the traditional "fsck" shown in Table 5.

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	122 sec	2.9 sec	0.025 sec
7.7Gb	591 sec	16.2 sec	0.050 sec

Table 7: *Background fsck times on an active filesystem*

Table 7 shows the times to run background "fsck" on a filesystem that has four processes concurrently writing to it. It is running with a *nice* value of four. Note that its running time increases by a factor of ten as it is yielding to the other running processes. By contrast, its effect on the other processes is minimal as their aggregate throughput is slowed by less than ten percent.

8. Conclusions and Future Work

This paper has described how to take snapshots of the fast filesystem with suspension intervals typically less than one second and independent of the size of the filesystem being snapshotted. When running with soft updates, the only filesystem corruption that occurs is the loss of inodes and data blocks. Using snapshots together with a slightly modified version of the traditional "fsck" it is possible to recover the lost inodes and data blocks while the filesystem is in active use. While a background "fsck" can run in about the same amount of time as a traditional "fsck", it is generally desirable to run it at a lower priority so that it causes less slowdown on other processes on the system.

Snapshots may seem a more complex solution to the problem of running space reclamation on an active filesystem than a more straight forward garbage-collection utility. However, their cost (about 1300 lines of code) is amortized over the other useful functionality: the ability to do reliable dumps of active filesystems and the ability to provide back-ups of the filesystem at several times during the day.

For the future, I need to gain experience with using background "fsck", to gain confidence in its robustness, and to find the optimal priority to minimize its slowdown on the system while still finishing its job in a reasonable amount of time.

9. Current Status

Snapshots and background "fsck" have been running on FreeBSD 5.0 systems since April 2001. All the relevant code including snapshots, the gating functions, the system call additions for the use of "fsck", and the changes to "fsck" itself are available as open-source under a Berkeley-style copyright.

10. Acknowledgments

I thank Rob Kolstad for his helpful comments and review of drafts of this paper. He pointed out several non-obvious gaps in the coverage. Thanks also to Matthew Dillon, Ian Dowse, Peter Wemm, and many other FreeBSD developers that put up with my kernel breakage and helped track down the bugs.

11. References

Ganger et al, 2000.

G. Ganger, M. McKusick, C. Soules, & Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in Filesystems," *ACM Transactions on Computer Systems,* 2, p. 127–153 (May 2000).

Hitz et al, 1994.

D. Hitz, J. Lau, & M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the San Francisco USENIX Conference,* p. 235–246 (January 1994).

Howard et al, 1988.

J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, & M. West, "Scale and Performance in a Distributed System," *ACM Transactions on Computer Systems,* 6, 1, p. 51–81 (February 1988).

Hutchinson et al, 1999.

N. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, & S. OMalley, "Logical vs. Physical File System Backup," *Third USENIX Symposium on Operating Systems Design and Implementation,* p. 239–250 (February 1999).

McKusick, et al., 1996.

M. McKusick, K. Bostic, M. Karels, & J. Quarterman,, *The Design and Implementation of the 4.4BSD Operating System,* p. 269–271, Addison Wesley Publishing Company, Reading, MA (1996).

McKusick & Ganger, 1999.

M. McKusick & G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Freenix Track at the Annual USENIX Technical Conference,* p. 1–17 (June 1999).

McKusick & Kowalski, 1994.

M. McKusick & T. Kowalski, "FSCK - The UNIX File System Check Program," *4.4 BSD System Manager's Manual,* p. 3:1–21, O'Reilley & Associates, Inc., Sebastopol, CA (April 1994).

Seltzer et al, 2000.

M. Seltzer, G. Ganger, M. McKusick, K. Smith, C. Soules, & C. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," *Proceedings of the San Diego USENIX Conference,* p. 71–84 (June 2000).

12. Biography

Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. His particular areas of interest are the virtual-memory system and the filesystem. He earned his undergraduate degree in Electrical Engineering from Cornell University, and did his graduate work at the University of California at Berkeley, where he earned Masters degrees in Computer Science and Business Administration, and a doctorate in Computer Science. He is a past president and present board member of the Usenix Association, and is a member of ACM and IEEE.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the web at http://www.mckusick.com/~mckusick/) in the basement of the house that he shares with Eric Allman, his domestic partner of 20-and-some-odd years. You can contact him via email at <mckusick@mckusick.com>.

Design And Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD

Kostas Magoutis
Division of Engineering and Applied Sciences, Harvard University
magoutis@eecs.harvard.edu

Abstract

The Direct Access File System (DAFS) is an emerging commercial standard for network-attached storage on server cluster interconnects. The DAFS architecture and protocol leverage network interface controller (NIC) support for user-level networking, remote direct memory access, efficient event notification, and reliable communication. This paper describes the design of the first implementation of a DAFS kernel server for FreeBSD, using existing interfaces with minor kernel modifications. We experimentally demonstrate that the current server structure can attain read throughput of more than 100 MB/s over a 1.25 Gb/s network even for small (i.e. 4K) block sizes when prefetching using an asynchronous client API. To reduce multithreading overhead and integrate the NIC with the host virtual memory system, our forthcoming system will incorporate new FreeBSD kernel support for asynchronous *vnode* I/O interfaces, integrating network and disk event notification and handling, and VM support for remote direct memory access. We believe our proposed kernel support is necessary to scale event-driven file servers to multi-gigabit network speeds.

1 Introduction

The emergence of network transports enabling low-overhead access to the network interface from user or kernel address space, remote direct memory access (RDMA), transport protocol offloading, and hardware support for event notification and delivery, has given rise to new applications and services that take advantage of these capabilities. The Direct Access File System [6] (DAFS) is a new commercial standard for file access over this new class of networks. DAFS grew out of the *DAFS Collaborative*, an industrial and academic consortium led by Network Appliance and Intel. DAFS file clients are usually applications that link with user-level libraries that implement the file access API. DAFS file servers are implemented in the kernel.

This paper describes the first implementation of a DAFS kernel server for FreeBSD. It is also as far as we know the first implementation in any general-purpose, demand-paged operating system. Section 2 summarizes the characteristics of network transports DAFS is designed to work with. Section 3.1 describes our prototype DAFS implementation using existing kernel interfaces. Section 3.2 introduces Optimistic DAFS, a new design we propose with potentially better performance but requiring additional kernel support. Section 4 outlines the kernel structure issues one is faced with in order to efficiently support DAFS kernel servers in FreeBSD. These include the need for asynchronous *vnode* interfaces to map and lock file pages in the buffer cache, integrating network and disk I/O event delivery, virtual memory support for network interface controller (NIC) memory management hardware, the need to revisit buffer cache locking assumptions, and a new BSD device driver model. Section 5 presents the performance of the current DAFS kernel prototype implementation and Section 6 summarizes our conclusions.

2 Memory-to-Memory Transports

DAFS [6] is a file access protocol specification deriving from NFS version 4 [22]. It is tailored for network transports (often referred to as *memory-to-memory* networks) providing user-level access to the network interface, remote direct memory access, efficient asynchronous event delivery mechanisms, and reliable communication semantics. Examples of memory-to-memory transports are Virtual Interface (VI) [5] and InfiniBand [11]. Current commercially available memory-to-memory network intercace have a long research heritage behind them [4, 23, 15]. The potential of advanced memory management features has also been consid-

ered [21, 24].

In this section we describe the characteristics of commercially available memory-to-memory networks that are relevant to a DAFS kernel server implementation.

Remote direct memory access (RDMA). Memory-to-memory networks are capable of data transfer between virtually addressed buffers in user process or kernel address space over the network. Hosts have to register virtual address mappings of buffers with the NIC prior to RDMA but are not involved in the actual data transfer. The programming interface to RDMA (except for buffer registration which is handled by the device driver) is usually through access to a memory-mapped data structure of transfer descriptors. Read, write (and sometimes atomic) remote memory access is allowed.

Registration of memory buffers with the NIC. The NIC includes a memory management unit in order to translate host virtual addresses to physical (bus) addresses to use in setting up DMA transfers. Most current commercially available NIC do not handle translation miss faults. The host needs to register (i.e. fill in mappings) with the NIC for all virtual memory regions the NIC is expected to access.

VM pages that have their mappings registered with the NIC have to be prevented from pageout at least while RDMA with them is in progress. Kernel interfaces that lock pages for the duration of an I/O suffice to prevent pageout when RDMA is locally initiated. With remotely initiated RDMA transfers that may happen at any time (as described in Section 3.2), only the NIC knows exactly when these transfers take place. To avoid excessive page locking by the host CPU, the NIC should have the ability to trigger or carry out page locking when needed. Support for integrating the NIC with the VM system is described in Section 4.3. Such support will enable a server to export large buffers (i.e. the entire VM cache) without underutilizing physical memory.

Efficient asynchronous event delivery mechanism. Memory-to-memory networks offer the *completion group* abstraction for scalable event notification and delivery. Completion groups simplify the task of simultaneously polling a large set of connections by aggregating their event notification and delivery into a single structure. Events such as receipt of a client request, or completion of a data transfer, can be efficiently detected and handled.

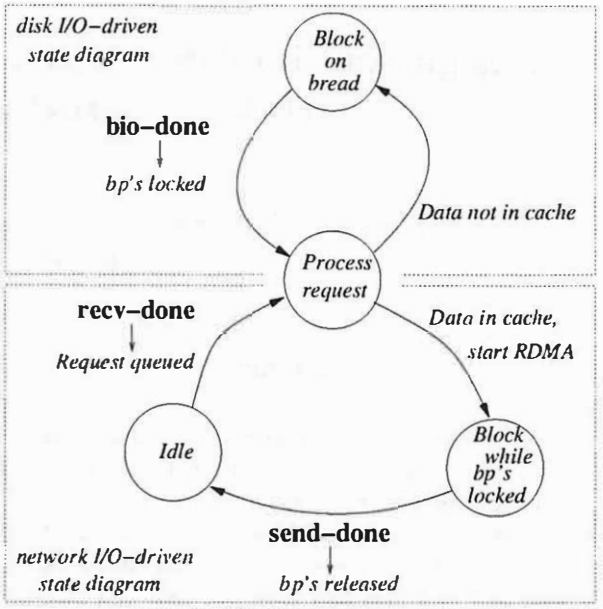

Figure 1: **Event-Driven DAFS Server.** Blocking is possible with existing interfaces.

Connection-oriented and reliable transport. Data transfer is usually over peer-to-peer transport connections (or channels). Reliable, exactly-once transport semantics are expected to be offered. Such semantics are usually implemented with hardware support in the network (as in the case of Fibre Channel [3]) or with end-to-end protocols implemented on the NIC (as in the case of VI/IP [9]). In either case, the host is not involved.

3 Direct Access File Systems

Direct access file systems are providing network file service over memory-to-memory transports. Section 3.1 introduces DAFS [6], an emerging commercial standard, and describes its prototype server implementation for FreeBSD. Section 3.2 introduces Optimistic DAFS, an improved design that relies more on RDMA and less on RPC for communication but requires special kernel support, particularly in the virtual memory subsystem.

3.1 DAFS

DAFS clients use lightweight RPC to communicate file requests to servers. In *direct* read or write operations the client provides virtual addresses of its

source or target memory buffers and data transfer is done using RDMA operations. RDMA operations are always issued by the server. In this paper we focus on server structure and I/O performance.

3.1.1 Server Design and Implementation

This section describes our current DAFS server design and implementation using existing FreeBSD kernel interfaces with minor kernel modifications. Our prototype DAFS kernel server follows the event-driven state transition diagram of Figure 1. Events are shown in boldface letters. The arrow under an event points to the action taken when the event occurs. The main events triggering state transitions are *recv-done* (a client-initiated transfer is done), *send-done* (a server-initiated transfer is done) and *bio-done* (a block I/O request from disk is done). Important design characteristics of the DAFS server in the current implementation are:

1. The server uses the buffer cache interface to do disk I/O (i.e. *bread()*, *bwrite()*, etc.). This is a zero-copy interface that can be used to lock buffers (pages and their mappings) for the duration of an RDMA transfer. RDMA transfers take place directly from or to the buffer cache.

2. RDMA transfers are initiated in the context of RPC handlers but proceed asynchronously. It is possible that an RDMA completes long after the RPC that initiated it has exited. Buffers involved in RDMA need to remain locked for the duration of the transfer. RDMA completion event handlers unlock those buffers and send an RPC reply if needed.

3. The kernel buffer cache manager is modified to register/deregister buffer mappings with the NIC on-the-fly, as physical pages are added or removed from buffers. This ensures that the NIC never takes translation miss faults and pages are wired only for the duration of the RDMA.

Each of the network and disk events has a corresponding event handler that executes in the context of a kernel thread.

1. *recv-done* is raised by the NIC and triggers processing of an incoming RPC request. For example, in the case of read or write operations the handler may initiate block I/O with the file system using *bread()*. After data is locked in buffers (hereafter referred to as *bp*'s) in the buffer cache, RDMA is initiated and the *bp*'s remain locked for the duration of the transfer.

2. *send-done* is raised by the NIC to notify of completion of a server-initiated (read or write) RDMA operation. The handler releases locks (using *brelse()*) on *bp*'s involved in the transfer and sends out an RPC response.

3. *bio-done* is raised by the disk controller and wakes up a thread that was blocking on disk I/O previously initiated by *bread()*. This event is currently handled by the kernel buffer cache manager in *biodone()*.

The server forks multiple threads to create concurrency in order to deal with blocking conditions. Kernel threads are created using an internal *rfork()* operation. One of the threads is responsible for listening for new transport connections while the rest are workers involved in data transfer. All transport connections are bound to the same completion group. Message arrivals on any transport connection generate *recv-done* interrupts which are routed to a single interrupt handler associated with the completion group. When the handler is invoked, it queues the incoming RPC request, notes the transport that was the source of the interrupt, and wakes up a worker thread to start processing. After parsing the request, a thread locks the necessary file pages in the buffer cache using *bread()*, prepares the RDMA descriptors and issues the RDMA operations. The RPC does not wait for RDMA completion. A later *send-done* interrupt (or successfull poll) on a completed RDMA transfer descriptor starts clean up and release of resources that the transfer was tying up (i.e. *bp* locks held on file buffers for the duration of the transfer), and sends out the RPC response. Threads blocking on those resources are awakened.

Event-driven design requires that event handlers be quick and not block between events. Our current server design deviates from this requirement due to the possibility of blocking under certain conditions:

1. Need to wait on disk I/O initiated by *bread()*. It is possible to avoid using the blocking *bread()* interface by initiating asynchronous I/O with the disk using *getblk()* followed by *strategy()*. We opted against this solution in our early design since disk event delivery is currently disjoint from network event delivery, complicating event handling. Integrating network and disk

I/O event delivery in the kernel is discussed in Sections 4.1 and 4.2.

2. Locking a *bp* twice by the same kernel thread or releasing a *bp* from a thread other than the lock owner causes a kernel panic (Section 4.4). Solutions are to (a) defer any request processing by a thread while past transfers it issued are still in progress, to ensure that a *bp* is always released by the lock owner and a thread never locks the same *bp* twice, or (b) modify the buffer cache so that these conditions no longer cause a kernel panic. To avoid wider kernel changes in the current implementation, we do (a). (b) is addressed in Section 4.4.

An important concern when designing an RDMA-based server is to minimize response latency for short transfers and maximize throughput for long transfers. In the current design, notification of incoming messages is done via interrupts and notification of server-initiated transfer completions via polling. Short transfers using RDMA are expected to complete within the context of their RPC request. In this way, the RPC response immediately follows RDMA completion, minimizing latency. Throughput is maximized for longer transfers by pipelining them as their RDMA operations can be concurrently progressing.

The low cost of DAFS RPC, the efficient event notification and delivery mechanism, and the absence of copies due to RDMA help towards low response latency. Maximum throughput is achievable even for small block sizes (as shown in Section 5) assuming the client is throwing requests at the server at a sufficiently high rate (i.e. doing prefetching using asynchronous I/O). The DAFS kernel server presently runs over the Emulex cLAN [8] and GN 9000 VI/IP [9] transports and is currently being ported to Mellanox InfiniBand [10].

3.2 Optimistic DAFS

In DAFS *direct* read and write operations, the client always uses an RPC to communicate the file access request along with memory references to client buffers that will be the source or target of a server-issued RDMA transfer. The cost associated with always having to do a file access RPC is manifested as unnecessarily high latency for small accesses from server memory. A way to reduce this latency is to allow clients to access the server file and VM cache directly rather than having to go each time through the server *vnode* interface via a file access RPC.

Optimistic DAFS [14] improves on the existing DAFS specification by reducing the number of file access RPC operations needed to initiate file I/O and replacing them with memory accesses using client-issued RDMA. Memory references to server buffers are given out to clients or other servers that maintain cache directories, and they are allowed to use those references to directly issue RDMA operations with server memory. To build cache directories, the server returns to the client a description of buffer locations in its VM cache (we assume a unified VM and file cache, as in FreeBSD). These buffer descriptions are returned either as a response to specific queries (i.e. client asks: *"give me the locations of all your resident pages associated with file foo"*), or piggybacked in the response to a read or write request (i.e. server responds: *"here's the data you asked for, and by the way, these are their memory locations that you can directly use in the future"*).

In Optimistic DAFS, clients use remote memory references found in their cache directories but accesses succeed only when directory entries have not become stale, for example as a result of actions of the server pageout daemon. There is no explicit notification to invalidate remote memory references previously given out on the network. Instead, remote memory access exceptions [14] thrown by the target NIC and caught by the initiator NIC can be used to discover invalid references and switch to the slower access path using file access RPC.

Maintaining the NIC memory management unit in the case where RDMA can be remotely initiated by a client at any time is tricky and needs special NIC and OS support. Section 4.3 describes the design of our forthcoming implementation that views the NIC as another processor in an asymmetric multiprocessor system and is based on the following design choices:

1. To make sure that exported pages have valid NIC mappings for as long as they are resident in physical memory and that these mappings are invalidated when pages are swaped to disk, paging activity on-the-fly adds or invalidates NIC mappings.

2. Being able to initiate DMA to and from main memory, the NIC (or the driver, in the absence of NIC support) has to synchronize and integrate with the VM system. To do that, it has to be able to manipulate *lock*, *reference*, and *dirty* bits of vm_pages in main memory.

3. To manage NIC mappings in servers with enormous physical memory sizes, the NIC address translation table is viewed as a cache of translations (i.e. a TLB). Translation misses are handled by the NIC (or the driver, in the absence of NIC support) and require access to page tables in main memory.

Previous research [21, 24] has looked at memory management of network interfaces but has not focused on kernel modifications or virtual memory system support. In Section 4.3 we address such support for the FreeBSD VM system. Finally, Optimistic DAFS requires maintainance of a directory on file clients (in user-space) and on other servers (in the kernel).

4 Kernel Support for DAFS Servers

Special capabilities and requirements of networking transports used by DAFS servers expose a number of kernel design and structure issues. In general, a DAFS file server needs to be able to

1. Do asynchronous file I/O
2. Integrate network and disk I/O event delivery
3. Lock file buffers while RDMA is in progress
4. Avoid memory copies

In what follows we describe our proposals for new kernel support in the FreeBSD kernel. Work on implementing these proposals is currently in progress. Section 4.1 argues for kernel asynchronous file I/O interfaces presently lacking in FreeBSD, and integrating network and file event notification and delivery. Section 4.2 presents a *vnode* interface designed to address these needs. Section 4.3 examines kernel support for memory management of the asymmetric multiprocessor system that consists of the NIC and the host CPU. Finally, Section 4.4 argues for modifications to buffer cache locking and Section 4.5 outlines device driver requirements of memory-to-memory NIC.

4.1 Event-Driven Design Support

An area of considerable interest in recent years [2, 18, 19] has been that of event-driven application design. Event-driven servers avoid much of the overhead associated with multithreaded designs but require truly asynchronous interfaces coupled with efficient event notification and delivery mechanisms integrating all types of events. The DAFS server requires such support in the kernel.

FreeBSD presently lacks an internal asynchronous interface to the buffer cache although it does provide an asynchronous I/O (AIO) system call API. AIO is an implementation of the POSIX 1003.1B standard and can be found in other systems such as Solaris [16]. It is implemented as a multithreaded interface to the filesystem over regular files, or using asynchronous device I/O over device-special files. The latter mechanism is more efficient as it avoids overhead associated with multithreading but can only be used for applications using raw device access, such as relational databases.

To integrate disk events with memory-to-memory NIC event handling, a generalization of the network-specific *completion group* abstraction is needed. Events from network and disk sources can be uniformly handled using *kqueue* [13], a recently introduced FreeBSD kernel abstraction for scalable event handling. *Kqueue* can be used to aggregate event sources such as a) completed network I/O, b) completed disk I/O, and c) process synchronization signals. A *kqueue* structure can handle all event types a DAFS server is interested in. Posting asynchronous operations requires registering *kevents* with the *kqueue*. Network and disk event handlers notify the server *kqueue* using appropriate event filters (i.e. EVFILT_KAIO for disk, and EVFILT_RDMA for network events). Notification can either be via polling (using *kqueue_scan()*) or, with a minor addition to the *kqueue* implementation, via a direct or delayed kernel upcall to a handler routine.

4.2 Vnode Interface Support

Vnode/VFS is a kernel interface that separates generic filesystem operations from specific filesystem implementations [20]. It was conceived to provide applications with transparent access to kernel filesystems, including network filesystem clients such as NFS. The *vnode*/VFS interface consists of two parts: VFS defines the operations that can be done on a filesystem. *Vnode* defines the operations that can be done on a file within a filesystem. Table 1 lists the *vnode* operations originally defined [20] to support NFS along with a number of local filesystems as well as later additions introduced into BSD systems with a unified file and VM cache to transfer data directly between the VM cache and the disk.

Existing *vnode* I/O interfaces are all syn-

Table 1: **Vnode ops (Sandberg et al.[20]).**

Vnode operation	Description
VOP_ACCESS	Check access permission
VOP_BMAP	Map block number
VOP_BREAD	Read a block
VOP_BRELSE	Release a block buffer
VOP_CLOSE	Mark file closed
VOP_CREATE	Create a file
VOP_FSYNC	Flush dirty blocks of a file
VOP_GETATTR	Return file attributes
VOP_INACTIVE	Mark *vnode* inactive
VOP_IOCTL	Do I/O control operation
VOP_LINK	Link to a file
VOP_LOOKUP	Lookup file name
VOP_MKDIR	Create a directory
VOP_OPEN	Mark file open
VOP_RDWR	Read or write a file
VOP_REMOVE	Remove a file
VOP_READLINK	Read symbolic link
VOP_RENAME	Rename a file
VOP_READDIR	Read directory entries
VOP_RMDIR	Remove directory
VOP_STRATEGY	Read/write fs blocks
VOP_SYMLINK	Create symbolic link
VOP_SELECT	Do select
VOP_SETATTR	Set file attributes
VOP_GETPAGES	Read and map pages in VM
VOP_PUTPAGES	Write mapped pages to disk

chronous. VOP_READ and VOP_WRITE take as an argument a `struct uio` buffer description and have copy semantics. VOP_GETPAGES and VOP_PUTPAGES are zero-copy interfaces transfering data directly between the VM cache and the disk. VM pages returned from VOP_GETPAGES need to be explicitly wired in physical memory to be used for device I/O. An interface for staging I/O should be designed to return buffers in a locked state. We believe that a *vnode* interface modeled after the low-level buffer cache interface with new support for asynchronous operation naturally fits the requirements of a DAFS server as outlined earlier. Such an asynchronous interface is easier to implement than an asynchronous version of VOP_GETPAGES, VOP_PUTPAGES, while being functionally equivalent to it in FreeBSD's unified VM and buffer cache.

Central to this new interface (summarized in Table 2) is a VOP_AREAD call which can be used to issue disk read requests and return without blocking. VOP_AREAD internally uses a new *aread()* buffer cache interface (described below) integrated with the *kqueue* mechanism. It takes as one of its arguments an asynchronous I/O control block (*kaiocb*) used to keep track of progress of the request.

```
aread(struct vnode *vp, struct kaiocb *cb)
{
    derive block I/O request from cb;
    bp = getblk(vp, block request);
    if (bp not found in the buffer cache) {
        register kevent using EVFILT_KAIO;
        register kaio_biodone handler with bp;
        VOP_STRATEGY(vp, bp);
    }
}
```

On completion of a request issued by *aread()*, the data is in a *bp*, in a locked state, and *kaio_biodone()* is called to deliver the event:

```
kaio_biodone(struct buf *bp)
{
    get kaiocb from bp;
    deliver event to knote in klist of kaiocb;
}
```

To unlock buffers and update filesystem state if necessary, VOP_BRELSE is used. Local filesystems would implement the interface of Table 2 in order to be efficiently exported by a DAFS server. For lack of this or another suitable interface, a local filesystem could always be exported by a DAFS server using existing interfaces, albeit with higher overhead mainly due to multithreading.

Network event delivery can be integrated with that of disk I/O as described earlier through the *kevent* mechanism. Each time an RDMA descriptor is issued, a *kevent* is registered using the EVFILT_RDMA filter and recorded in the completion group (CG) structure. Completion group handlers need to deal with *kqueue* event delivery:

```
send_event(CG *cq, Transport *vi)
{
    deliver event to knote in klist of CG;
}
```

The DAFS server is notified of new events by periodically polling the *kqueue*. Alternatively, a common handler is invoked each time a network or disk event occurs.

Table 2: **Vnode Interface to Buffer Cache.**

Vnode operation	Description
VOP_BREAD	Lock all buffers needed for I/O; read from *vp*.
VOP_AREAD	Lock all buffers needed for I/O; read from *vp*; don't block .
VOP_BDWRITE	Mark dirty entries; delayed write to *vp*; update state if requested.
VOP_BWRITE	Block writing to *vp*; update state if requested.
VOP_BAWRITE	Mark dirty entries; async write to *vp*; update state if requested.
VOP_BRELSE	Unlock buffers; update file state if requested.

We illustrate the use of the proposed *vnode* interface to the buffer cache by breaking down and describing the steps in *read* and *write* operations implemented by a DAFS server. For comparison with existing interfaces, we describe the same steps implemented by NFS. Without loss of generality we assume an FFS underlying filesystem at the server.

Read. With DAFS, a client (direct) read request carries the remote memory address of the client buffers. The DAFS server issues a VOP_AREAD to read and lock all necessary file blocks in the buffer cache. VOP_AREAD starts disk operations and returns without blocking, after registering with *kqueue*. When pages are resident and locked and the server notified via *kqueue*, it issues RDMA Write operations to client memory for all requested file blocks. When the transfers are done, the server does VOP_BRELSE to unlock the file buffer cache blocks.

With NFS, on a client read operation the server issues a VOP_READ to the underlying filesystem with a `uio` parameter pointing to a gather/scatter list of mbufs that will eventually form the response to the read request RPC. In the FFS implementation of VOP_READ and without applying any optimizations, a loop reads and locks file blocks into the buffer cache using *bread()*, subsequently copying the data into the mbufs pointed to by `uio`. For page-aligned, page-sized buffers, page-flipping techniques can be applied to save the copy into the mbufs.

Write. With DAFS, a client (direct) write request carries only client memory addresses of data buffers. The DAFS server uses VOP_AREAD to read and lock in the buffer cache all necessary file blocks. When pages are resident and locked, it issues RDMA Read requests to fetch the data from the client buffers directly into the buffer cache blocks. When the transfer is done, the server uses one of VOP_BWRITE, VOP_BDWRITE, VOP_BAWRITE, depending on whether this is a stable write request or not, to issue a disk write I/O and unlock the buffers. Additionally, a metadata update is effected if requested.

With NFS, a client write operation carries the data to be written inline with the RPC request. The NFS server prepares a `uio` with a gather/scatter list of all the data mbufs and calls VOP_WRITE. Apart from the `uio` parameter that describes the transfer, an `ioflags` parameter is passed signifying whether the write to disk should happen synchronously. With NFS version 2 all writes and metadata updates are synchronous. NFS versions 3 and 4 allow asynchronous writes. In the FFS implementation of VOP_WRITE, a loop reads and locks the file blocks to be written into the buffer cache using *bread()*, copies into them the data described by `uio`, then uses one of *bwrite* (synchronous), *bdwrite* (delayed), or *bawrite* (asynchronous) writes depending on whether this is a stable write request (see `ioflags`) or not. Finally, a metadata update is effected if requested.

An interesting note on the ability of the DAFS server to implement file writes using RDMA Read (instead of client-initiated RPC or RDMA Write) is that this enables it to read data from client memory no faster than dirty buffers can be written to disk. This *bandwidth matching* capability becomes very important in multi-gigabit networks when network bandwidth is often greater than disk bandwidth.

4.3 VM System Support

In systems deriving from 4.4BSD [17], maintaing virtual/physical address mappings and page access rights used by the main CPU memory-management hardware is done by the machine-dependent *physical mapping (pmap)* module. Low level machine-independent kernel code such as the buffer cache, kernel malloc and the rest of the VM system are using *pmap* to add or remove address mappings and alter page access rights. Symmetric multiprocessor (SMP) systems sharing main mem-

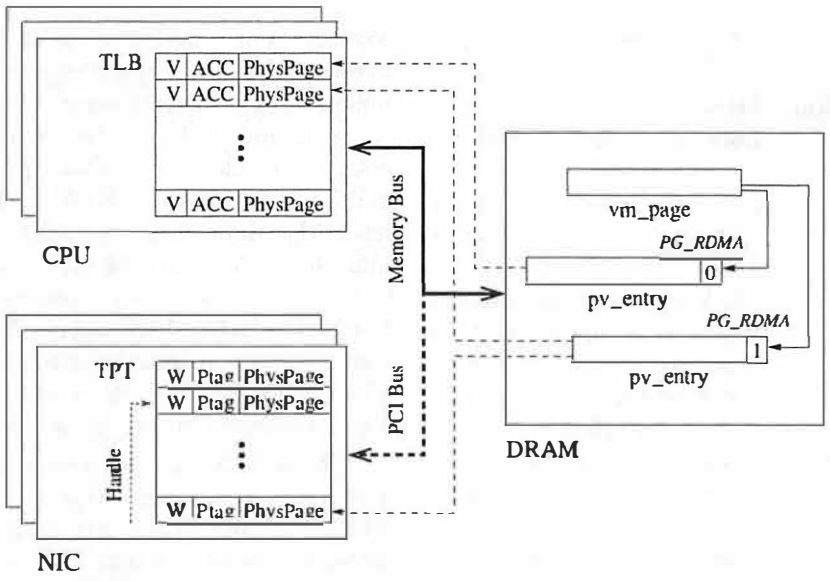

Figure 2: **CPU and NIC Memory Management Hardware.**

ory can use a single *pmap* module as long as translation lookaside buffers (TLB) on each CPU are kept consistent. *Pmap* operations apply to page tables shared by all CPU. TLB miss exceptions thrown by a CPU result in a lookup for mappings in the shared page tables. Invalidations of mappings are applied to all CPU.

Memory-to-memory NIC store virtual-to-physical address translations and access rights for all user and kernel memory regions directly addressable and accessible by the NIC. Figure 2 shows a system combining both CPU and NIC memory management hardware: Main CPU use their on-chip translation lookaside buffer (TLB) to translate virtual to physical addresses. A typical TLB page entry includes a number of bits such as V and ACC signifying whether the page translation is valid, and what the access rights to the page are, along with the physical page number. A miss on a TLB lookup requires a page table lookup in main memory.

NIC on the PCI (or other I/O) bus have their own translation and protection (TPT) [5] tables. Each entry in the TPT includes bits enabling RDMA Read or Write (i.e. the W bit in the diagram) operations on the page, the physical page number, and a Ptag value identifying the process that owns the pages (or the kernel). Whereas the TLB is a high-speed associative memory, the TPT is usually implemented as a DRAM module on the NIC board. To accelerate lookups on the TPT, remote memory access requests carry a Handle index that helps the NIC find the right TPT entry.

This section focuses on operating system support to integrate NIC memory management units in the FreeBSD VM system. The main benefits of this integration are

1. VM pages exported for RDMA are wired in physical memory only for as long as RDMA transfers from or to them are in progress, resulting in better utilization of physical memory.

2. The entire VM cache (i.e. potentially all file VM objects) can be safely exported in the face of paging activity.

We consider the NIC as a processor (with special I/O capabilities [12]) in the asymmetric multiprocessor system of Figure 2 and allow sharing of kernel VM structures in main memory between NIC and main processors. We assume that access to VM structures on behalf of the NIC is done by the CPU, executing driver handlers in response to interrupts. Direct access by the NIC to VM structures in main memory is considered later in this section.

In FreeBSD (and other systems deriving from 4.4BSD), a physical page is represented by a `vm_page` structure and an address space by a `vm_map` structure. A page may be mapped on one or more `vm_maps` with each mapping represented by a `pv_entry` structure. Figure 2 shows a `vm_page` with two associated `pv_entry` structures in main memory.

Table 3: **Low-level NIC VM primitives.**

Function	Description
tpt_init()	Initialize
tpt_enter()	Insert mapping
tpt_remove()	Remove mapping
tpt_protect()	Protect mapping

Table 4: **VM interface used by the NIC.**

Function	Description
vm_page_io_start()	Lock page
vm_page_io_finish()	Unlock page
vm_nic_fault()	Handle NIC fault
vm_page_reference()	Reference page
vm_page_dirty()	Dirty page

In our design, the VM system maintains the NIC MMU via the OS-NIC interface. The NIC accesses VM structures via the NIC-OS interface.

OS-NIC Interface. The OS needs to interact with the NIC to add, remove and modify VM mappings stored on its TPT. A mapping of a VM page expected to be used in RDMA has to be registered with the NIC. Registering the mapping with the NIC happens in *pmap*, right after the CPU mapping with a `vm_map` is established. The NIC exports low-level VM primitives (Table 3) for use by *pmap* to add, remove and modify TPT entries. NIC mappings may later be deregistered (when the original mapping is removed/invalidated), or have their protection changed.

To keep an account of VM mappings that have been registered with the NIC, we add a PG_RDMA bit in the `pv_entry` structure to be set whenever a `pv_entry` has a NIC as well as a CPU mapping. In Figure 2, the `pv_entry` with the PG_RDMA bit set has both a CPU and a NIC mapping. In all *pmap* operations on VM pages, the *pmap* module interacts with the NIC only if the PG_RDMA flag is set on the `pv_entry`.

Higher-level code can trigger registration of a virtual memory region with the NIC by propagating appropriate flags from higher to lower-level interfaces and eventually to the *pmap*. For example, the DAFS server sets an IO_RDMA bit in the `ioflags` parameter of the *vnode* interface (Table 2) when planning to use the buffer for RDMA. This eventually translates into a VM_RDMA flag in the *pmap_enter()* interface that results in mappings being registered with the NIC.

A problem with invalidating `pv_entry` mappings that have also been registered with the NIC is that NIC invalidations may need to be delayed for as long as RDMA transfers using the mappings are in progress. *Pmap_remove_all()* is complicated by this fact as (for atomicity) it has to try to remove `pv_entry` structures with NIC mappings first and may eventually fail if NIC invalidations are not possible within a reasonable amount of time.

Another problem is with VM system policies that are often based on architectural assumptions that do not hold with NIC characteristics. For example, the FreeBSD VM system unmaps pages from process page tables when moving them from an active to inactive or cached state. This is because the VM system is willing to take a reasonable number of reactivation faults to determine how active a page actually is, based on the assumption that reactivation faults are relatively inexpensive [7]. NIC reactivation faults are significantly more expensive compared to CPU faults due to lack of integration between the NIC and the host memory system. To reduce that cost, it would make sense to apply the deactivation policy only to CPU mappings, leaving NIC mappings intact for as long as VM pages are memory resident. However, full integration of the NIC memory management unit into the VM system argues for this policy to be equally applied to NIC page accesses.

NIC-OS Interface. The NIC initiates interaction with the VM system in the following occasions using the interface of Table 4.

1. The NIC can be the initiator of DMA from or to VM pages in main memory for incoming RDMA Read or Write requests, and thus has to be able to lock (busy) VM pages for the duration of the transfer. The interface is similar to the kernel interface used to busy pages during pagein or pageout activity.

2. On a translation miss, the NIC needs to do a page table lookup for the missed virtual address. A new translation is loaded in the NIC TPT if the page is found to be resident in memory. If not, pagein may be initiated by the miss handler but the NIC may choose not to wait for it and report an RDMA exception instead.

3. The NIC updates *reference*, *dirty* bits whenever it accesses or writes to VM pages.

All this handling can be done by the host CPU in response to interrupts thrown by the NIC. For efficient access to vm_page bits in main memory without interrupting the host CPU, the NIC should share definitions of VM structures and store direct references to vm_pages. These references could either be physical (bus) addresses, or virtual addresses that are translated using the NIC TPT. In cases where a simple bit flip on a vm_page is needed, the NIC should be able to do that by a direct atomic memory access. Complicated page table lookups (i.e. in translation miss handling) are better handled by the host CPU.

4.4 Buffer Cache Locking

In our first implementation of a DAFS server, we chose to directly use the buffer cache for block I/O. In an RDMA-based data transfer, the server sets up the RDMA transfer in the context of the requesting RPC. Once issued, the RDMA proceeds asynchronously to the RPC. The latter does not wait for RDMA completion. To serialize concurrent access to shared files in the face of asynchrony, the *vnode* (*vp*) of a file needs to be locked for the duration of the RPC. However, the data buffers (*bp*'s) transfered need to be locked for the full duration of the RDMA. Locking the *vp* (i.e. the entire file) for the duration of the RDMA would also work but would limit performace in case of sharing since requests for non-overlapping regions of a file would have to serialize. Our decision to lock at a finer granularity than the *vp* for the duration of a transfer conflicts with current FreeBSD buffer cache locking assumptions:

1. Locking a buffer in the cache requires a process to acquire an exclusive lock on that buffer. A buffer lock can only be released by the same process that locked it or by the kernel.

2. Before an asynchronous disk I/O (i.e. an asynchronous write, or readahead), lock ownership has to be transfered to the kernel so that the block can later be released by the kernel (in *biodone()*).

A multithreaded event-driven kernel server that directly uses the buffer cache and does event processing in kernel process context faces problems in the following circumstances:

1. When a thread tries to lock a buffer it is already locking (because a transfer is in progress on that buffer) expecting to block until that lock is released by some other thread.

2. When a buffer is released from a different thread than the one that locked it.

Transfering lock ownership to the kernel during asynchronous network I/O does not help since lock release is done by some kernel process (whichever happens to have polled for that particular event) rather than by the kernel itself. The solution presently used is for the kernel process that issued an RDMA operation to wait until the transfer is done in order to release the lock. This also prohibits that process from trying to lock the same buffer again, thus causing a deadlock panic. A better solution is to enable recursive locking and allow lock release by any of the server threads.

4.5 Device Driver Support

Memory-to-memory network adapters virtualize the NIC hardware and are directly accessible from user space. One such example is VI [5] where the NIC implements a number of VI contexts. Each VI is the equivalent of a socket in traditional network protocols, except that a VI is directly supported by the NIC hardware and usually has a memory-mapped rather than a system call interface. The requirement to create multiple logical instances of a device, each with its own private state (separate from the usual device softcopy state) and to map those devices in user address spaces requires new support from BSD kernels.

Network driver model. Network drivers in BSD systems are traditionally accessed through sockets and do not appear in the filesystem name space (i.e. under /dev). User-level libraries for memory-to-memory network transports require these devices to be opened and closed multiple times with each opened instance appearing as a separate logical device maintaining private state, and be memory-mapped. FreeBSD until recently provided rudimentary support for this through the *devfs* file system which is being abandoned. *Devfs* allows logical instances of a device to be dynamically created but still associates a single *vnode* with that device. Our current drivers rely on a hack to associate a separate *vnode* and store private state with each logical instance of a device.

Device driver models under other operating

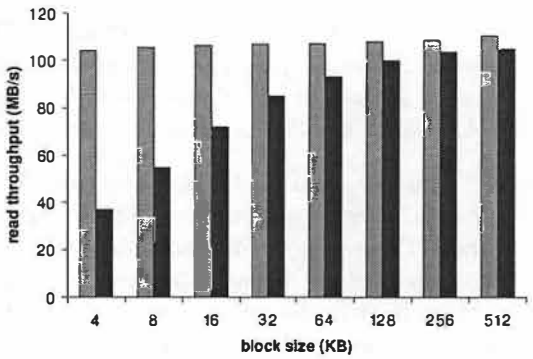

Figure 3: **DAFS Read Throughput.** From left to right: asynchronous API; synchronous API.

systems have different ways for logical device instances to maintain private state. Linux associates a *vnode* with each opened instance of a device file, and Solaris keeps private per-instance state via DDI [1].

5 Performance

We evaluate performance of the prototype DAFS server with a synthetic benchmark that involves a client fetching random blocks from warm server cache (no disk I/O). We measured performance for both the synchronous and asynchronous API. In the asynchronous case, the client issues read I/O maintaining up to 64 outstanding requests at any time. The server was configured to run with 64 kernel threads. In the synchronous case, the client was blocking waiting for completion on each request.

The experiment runs over two Pentium III 800 MHz systems with the ServerWorks LE chipset and 1GB RAM, connected via 1.25 Gb/s cLAN [8] VI cards on 64-bit/66MHz PCI over a cLAN switch. The network has been measured to yield 113 MB/s with a VI one-way throughput benchmark (using 32K packets and polling) provided by Giganet. We measured the effect of the block size used in read requests varying it from 4KB to 512KB and report results in Figure 3. We see that even for small block sizes, performance using the asynchronous interface is close to wire throughput by being able to pipeline server responses. Low DAFS overhead and the absence of copying reduces the memory bottleneck that would otherwise be a limiting factor. Performance using the synchronous interface converges to almost wire throughput for block sizes of about 128KB when the per-I/O overhead is fully amortized.

6 Conclusions

This paper focused on the kernel issues involved in building Direct Access File System servers. A range of issues was addressed drawing from our experience in building such a kernel server for FreeBSD. We described the current server structure using existing interfaces with minor kernel modifications. Performance results show that our current DAFS server prototype implementation can offer high performance file service for memory workloads over a 1.25 Gb/s network.

A problem with existing blocking kernel interfaces is that DAFS and other kernel servers using them experience the overhead of having to associate a process context with each I/O request. This overhead is expected to become more pronounced in multi-gigabit networks. Our proposed additions to the *vnode* interface offer support for asynchronous file I/O with integrated network and disk event notification and delivery.

We presented design possibilities for integrating a programmable RDMA-capable NIC with the FreeBSD VM system. This support will allow a DAFS server to export its entire VM cache over the network in the face of client-initiated RDMA operations and server paging activity. We are currently planning an implementation that embodies such a design and can be used to support Optimistic DAFS.

Finally, we have found that the existing BSD network driver model is inadequate to support the needs of memory-to-memory NIC devices and a new model is needed.

7 Acknowledgments

The author would like to thank Donn Seeley, the shepherd for this paper, and Alexandra Fedorova and Salimah Addetia for helpful feedback.

8 Software Status and Availability

The current prototype runs as a kernel loadable module for FreeBSD 4.3-RELEASE and implements a large subset of the DAFS specification including all basic file access, performance enhancements, locking, and client caching support operations. Besides the server, we have ported device drivers to FreeBSD for a number of VI NIC, including the Giganet/Emulex cLAN 1000 and GN 9000 VI/IP which use ATM and gigabit Ethernet respec-

tively as their link layer transport. We anticipate to measure the performance benefits of Optimistic DAFS as soon as the transport support is implemented. Source code for the server and associated FreeBSD kernel patches can be obtained from `http://www.eecs.harvard.edu/vino/fs-perf/dafs`.

References

[1] *Writing Device Drivers.* Sun Microsystems, Inc., 2000.

[2] G. Banga et al. Better Operating System Features for Faster Network Servers. In *Proceedings of the Workshop on Internet Server Performance (WISP)*, June 1998.

[3] A. Benner. *Fibre Channel: Gigabit I/O and Communications for Computer Networks.* McGraw-Hill, 1996.

[4] M. Blumrich et al. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[5] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.

[6] DAFS Collaborative. *Direct Access File System Protocol, Version 1.0*, September 2001. `http://www.dafscollaborative.org`.

[7] M. Dillon. Design Elements of the FreeBSD VM System. `http://www.daemonnews.org/200001/freebsd_vm.html`.

[8] Emulex/Giganet Inc. cLAN 1000 VI adapter. `http://wwwip.emulex.com`.

[9] Emulex/Giganet Inc. GN 9000 VI adapter. `http://wwwip.emulex.com`.

[10] Mellanox Inc. Sleek 1X InfiniBand Channel Adapter. `http://www.mellanox.com`.

[11] InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.0*, October 2000.

[12] Intel Inc. I/O Processors. `http://developer.intel.com/design/iio/80310.htm`.

[13] J. Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *2001 USENIX Technical Conference - FREENIX Track*, June 2001.

[14] K. Magoutis. The Optimistic Direct Access File System. *Submitted for publication at the Workshop on Novel Uses of System Area Networks (SAN 2001), Cambridge, MA*, February 2002.

[15] E.P. Markatos and M. G. H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture, San Jose, CA*, pages 144–153, February 1996.

[16] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture.* Prentice Hall, 2000.

[17] M. Kirk McKusick et al. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[18] J. Ousterhout. Why Threads are a Bad Idea (For Most Purposes). Invited Talk at the 1996 USENIX Technical Conference, January 1996.

[19] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *1999 USENIX Technical Conference*, June 1999.

[20] R. Sandberg et al. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Technical Conference, Portland, OR*, pages 119–130, June 1985.

[21] I. Schoinas and M. D. Hill. Address Translation Mechanisms in Network Interfaces. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA)*, February 1998.

[22] S. Shepler et al. NFS Version 4 Protocol. RFC 3010, December 2000.

[23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[24] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of the 1997 Hot Interconnects Symposium*, August 1997.

Rethinking /dev and devices in the UNIX kernel

Poul-Henning Kamp
<phk@FreeBSD.org>
The FreeBSD Project

Abstract

An outstanding novelty in UNIX at its introduction was the notion of "a file is a file is a file and even a device is a file." Going from "hardware only changes when the DEC Field engineer is here" to "my toaster has USB" has put serious strain on the rather crude implementation of the "devices as files" concept, an implementation which has survived practically unchanged for 30 years in most UNIX variants. Starting from a high-level view of devices and the semantics that have grown around them over the years, this paper takes the audience on a grand tour of the redesigned FreeBSD device-I/O system, to convey an overview of how it all fits together, and to explain why things ended up as they did, how to use the new features and in particular how not to.

1. Introduction

There are really only two fundamental ways to conceptualise I/O devices in an operating system: The usual way and the UNIX way.

The usual way is to treat I/O devices as their own class of things, possibly several classes of things, and provide APIs tailored to the semantics of the devices. In practice this means that a program must know what it is dealing with, it has to interact with disks one way, tapes another and rodents yet a third way, all of which are different from how it interacts with a plain disk file.

The UNIX way has never been described better than in the very first paper published on UNIX by Ritchie and Thompson [Ritchie74]:

> Special files constitute the most unusual feature of the UNIX filesystem. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory /dev, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file /dev/mt.
>
> Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special files are protected from indiscriminate access.
>
> There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

At the time, this was quite a strange concept; it was totally accepted for instance, that neither the system administrator nor the users were able to interact with a disk as a disk. Operating systems simply did not provide access to disk other than as a filesystem. Most vendors did not even release a program to initialise a disk-pack with a filesystem: selling pre-initialised and "quality tested" disk-packs was quite a profitable business.

In many cases some kind of API for reading and writing individual sectors on a disk pack did exist in the operating system, but more often than not it was not listed in the public documentation.

1.1. The traditional implementation

The initial implementation used hardcoded inode numbers [Ritchie98]. The console device would be inode number 5, the paper-tape-punch number 6 and so on, even if those inodes were also actual regular files in the filesystem.

For reasons one can only too vividly imagine, this was changed and Thompson [Thompson78] describes how the implementation now used "major and minor" device numbers to index though the devsw array to the correct device driver.

For all intents and purposes, this is the implementation which survives in most UNIX-like systems even to this day. Apart from the access control and timestamp information which is found in all inodes, the special

inodes in the filesystem contain only one piece of information: the major and minor device numbers, often logically OR'ed to one field.

When a program opens a special file, the kernel uses the major number to find the entry points in the device driver, and passes the combined major and minor numbers as a parameter to the device driver.

2. The challenge

Now, we did not talk much about where the special inodes came from to begin with. They were created by hand, using the mknod(2) system call, usually through the mknod(8) program.

In those days a computer had a very static hardware configuration[1] and it certainly did not change while the system was up and running, so creating device nodes by hand was certainly an acceptable solution.

The first sign that this would not hold up as a solution came with the advent of TCP/IP and the telnet(1) program, or more precisely with the telnetd(8) daemon. In order to support remote login a "pseudo-tty" device driver was implemented, basically as tty driver which instead of hardware had another device which would allow a process to "act as hardware" for the tty. The telnetd(8) daemon would read and write data on the "master" side of the pseudo-tty and the user would be running on the "slave" side, which would act just like any other tty: you could change the erase character if you wanted to and all the signals and all that stuff worked.

Obviously with a device requiring no hardware, you can compile as many instances into the kernel as you like, as long as you do not use too much memory. As system after system was connected to the ARPANet, "increasing number of ptys" became a regular task for system administrators, and part of this task was to create more special nodes in the filesystem.

Several UNIX vendors also noticed an issue when they sold minicomputers in many different configurations: explaining to system administrators just which special nodes they would need and how to create them were a significant documentation hassle. Some opted for the simple solution and pre-populated /dev with every conceivable device node, resulting in a predictable slowdown on access to filenames in /dev.

System V UNIX provided a band-aid solution: a special boot sequence would take effect if the kernel or the hardware had changed since last reboot. This boot procedure would amongst other things create the necessary special files in the filesystem, based on an intricate system of per device driver configuration files.

In the recent years, we have become used to hardware which changes configuration at any time: people plug USB, Firewire and PCCard devices into their computers. These devices can be anything from modems and disks to GPS receivers and fingerprint authentication hardware. Suddenly maintaining the correct set of special devices in "/dev" became a major headache.

Along the way, UNIX kernels had learned to deal with multiple filesystem types [Heidemann91a] and a "device-pseudo-filesystem" was a pretty obvious idea. The device drivers have a pretty good idea which devices they have found in the configuration, so all that is needed is to present this information as a filesystem filled with just the right special files. Experience has shown that this like most other "pseudo filesystems" sound a lot simpler in theory than in practice.

3. Truly understanding devices

Before we continue, we need to fully understand the "device special file" in UNIX.

First we need to realize that a special file has the nature of a pointer from the filesystem into a different namespace; a little understood fact with far reaching consequences.

One implication of this is that several special files can exist in the filename namespace all pointing to the same device but each having their own access and timestamp attributes:

```
guest# ls -l /dev/fd0 /tmp/fd0
crw-r----- 1 root operator 9, 0 Sep 27 19:21 /dev/fd0
crw-rw-rw- 1 root wheel    9, 0 Sep 27 19:24 /tmp/fd0
```

Obviously, the administrator needs to be on top of this: one popular way to exploit an unguarded root prompt is to create a replica of the special file /dev/kmem in a location where it will not be noticed. Since /dev/kmem gives access to the kernel memory, gaining any particular privilege can be arranged by suitably modifying the kernel's data structures through the illicit special file.

When NFS appeared it opened a new avenue for this attack: People may have root privilege on one machine but not another. Since device nodes are not interpreted on the NFS server but rather on the local computer, a user with root privilege on a NFS client computer can create a device node to his liking on a filesystem mounted from an NFS server. This device node can in turn be used to circumvent the security of other computers which mount that filesystem, including the

[1] Unless your assigned field engineer was present on site.

server, unless they protect themselves by not trusting any device entries on untrusted filesystem by mounting such filesystems with the nodev mount-option.

The fact that the device itself does not actually exist inside the filesystem which holds the special file makes it possible to perform boot-strapping stunts in the spirit of Baron Von Münchausen [raspe1785], where a filesystem is (re)mounted using one of its own device vnodes:

```
guest# mount -o ro /dev/fd0 /mnt
guest# fsck /mnt/dev/fd0
guest# mount -u -o rw /mnt/dev/fd0 /mnt
```

Other interesting details are chroot(2) and jail(2) [Kamp2000] which provide filesystem isolation for process-trees. Whereas chroot(2) was not implemented as a security tool [Mckusick1999] (although it has been widely used as such), the jail(2) security facility in FreeBSD provides a pretty convincing "virtual machine" where even the root privilege is isolated and restricted to the designated area of the machine. Obviously chroot(2) and jail(2) may require access to a well-defined subset of devices like /dev/null, /dev/zero and /dev/tty, whereas access to other devices such as /dev/kmem or any disks could be used to compromise the integrity of the jail(2) confinement.

For a long time FreeBSD, like almost all UNIX-like systems had two kinds of devices, "block" and "character" special files, the difference being that "block" devices would provide caching and alignment for disk device access. This was one of those minor architectural mistakes which took forever to correct.

The argument that block devices were a mistake is really very very simple: Many devices other than disks have multiple modes of access which you select by choosing which special file to use.

Pick any old timer and he will be able to recite painful sagas about the crucial difference between the /dev/rmt and /dev/nrmt devices for tape access.[2]

Tapes, asynchronous ports, line printer ports and many other devices have implemented submodes, selectable by the user at a special filename level, but that has not earned them their own special file types. Only disks[3] have enjoyed the privilege of getting an entire file type dedicated to a a minor device mode.

Caching and alignment modes should have been enabled by setting some bit in the minor device number on the disk special file, not by polluting the filesystem code with another file type.

In FreeBSD block devices were not even implemented in a fashion which would be of any use, since any write errors would never be reported to the writing process. For this reason, and since no applications were found to be in existence which relied on block devices and since historical usage was indeed historical [Mckusick2000], block devices were removed from the FreeBSD system. This greatly simlified the task of keeping track of open(2) reference counts for disks and removed much magic special-case code throughout.

4. Files, sockets, pipes, SVID IPC and devices

It is an instructive lesson in inconsistency to look at the various types of "things" a process can access in UNIX-like systems today.

First there are normal files, which are our reference yardstick here: they are accessed with open(2), read(2), write(2), mmap(2), close(2) and various other auxiliary system calls.

Sockets and pipes are also accessed via file handles but each has its own namespace. That means you cannot open(2) a socket,[4] but you can read(2) and write(2) to it. Sockets and pipes vector off at the file descriptor level and do not get in touch with the vnode based part of the kernel at all.

Devices land somewhere in the middle between pipes and sockets on one side and normal files on the other. They use the filesystem namespace, are implemented with vnodes, and can be operated on like normal files, but don't actually live in the filesystem.

Devices are in fact special-cased all the way through the vnode system. For one thing devices break the "one file-one vnode" rule, making it necessary to chain all vnodes for the same device together in order to be able to find "the canonical vnode for this device node", but more importantly, many operations have to be specifically denied on special file vnodes since they do not make any sense.

For true inconsistency, consider the SVID IPC mechanisms - not only do they not operate via file handles, but they also sport a singularly illconceived 32 bit numeric namespace and a dedicated set of system calls for access.

[2] Make absolutely sure you know the difference before you take important data on a multi-file 9-track tape to remote locations.

[3] Well, OK: and some 9-track tapes.

[4] This is particularly bizarre in the case of UNIX domain sockets which use the filesystem as their namespace and appear in directory listings.

Several people have convincingly argued that this is an inconsistent mess, and have proposed and implemented more consistent operating systems like the Plan9 from Bell Labs [Pike90a] [Pike92a]. Unfortunately reality is that people are not interested in learning a new operating system when the one they have is pretty darn good, and consequently research into better and more consistent ways is a pretty frustrating [Pike2000] but by no means irrelevant topic.

5. Solving the /dev maintenance problem

There are a number of obvious, simple but wrong ways one could go about solving the "/dev" maintenance problem.

The very straightforward way is to hack the namei() kernel function responsible for filename translation and lookup. It is only a minor matter of programming to add code to special-case any lookup which ends up in "/dev". But this leads to problems: in the case of chroot(2) or jail(2), the administrator will want to present only a subset of the available devices in "/dev", so some kind of state will have to be kept per chroot(2)/jail(2) about which devices are visible and which devices are hidden, but no obvious location for this information is available in the absence of a mount data structure.

It also leads to some unpleasant issues because of the fact that "/dev/foo" is a synthesised directory entry which may or may not actually be present on the filesystem which seems to provide "/dev". The vnodes either have to belong to a filesystem or they must be special-cased throughout the vnode layer of the kernel.

Finally there is the simple matter of generality: hard-coding the string "/dev" in the kernel is very general.

A cruder solution is to leave it to a daemon: make a special device driver, have a daemon read messages from it and create and destroy nodes in "/dev" in response to these messages.

The main drawback to this idea is that now we have added IPC to the mix introducing new and interesting race conditions.

Otherwise this solution is a surprisingly effective, but chroot(2)/jail(2) requirements prevents a simple implementation and running a daemon per jail would become an administrative nightmare.

Another pitfall of this approach is that we are not able to remount the root filesystem read-write at boot until we have a device node for the root device, but if this node is missing we cannot create it with a daemon since the root filesystem (and hence /dev) is read-only. Adding a read-write memory-filesystem mount /dev to solve this problem does not improve the architectural qualities further and certainly the KISS principle has been violated by now.

The final and in the end only satisfactory solution is to write a "DEVFS" which mounts on "/dev".

The good news is that it does solve the problem with chroot(2) and jail(2): just mount a DEVFS instance on the "dev" directory inside the filesystem subtree where the chroot or jail lives. Having a mountpoint gives us a convenient place to keep track of the local state of this DEVFS mount.

The bad news is that it takes a lot of cleanup and care to implement a DEVFS into a UNIX kernel.

6. DEVFS architectural decisions

Before implementing a DEVFS, it is necessary to decide on a range of corner cases in behaviour, and some of these choices have proved surprisingly hard to settle for the FreeBSD project.

6.1. The "persistence" issue

When DEVFS in FreeBSD was initially presented at a BoF at the 1995 USENIX Technical Conference in New Orleans, a group of people demanded that it provide "persistence" for administrative changes.

When trying to get a definition of "persistence", people can generally agree that if the administrator changes the access control bits of a device node, they want that mode to survive across reboots.

Once more tricky examples of the sort of manipulations one can do on special files are proposed, people rapidly disagree about what should be supported and what should not.

For instance, imagine a system with one floppy drive which appears in DEVFS as "/dev/fd0". Now the administrator, in order to get some badly written software to run, links this to "/dev/fd1":

```
ln /dev/fd0 /dev/fd1
```

This works as expected and with persistence in DEVFS, the link is still there after a reboot. But what if after a reboot another floppy drive has been connected to the system? This drive would naturally have the name "/dev/fd1", but this name is now occupied by the administrators hard link. Should the link be broken? Should the new floppy drive be called "/dev/fd2"? Nobody can agree on anything but the ugliness of the situation.

Given that we are no longer dependent on DEC Field engineers to change all four wheels to see which one is

flat, the basic assumption that the machine has a constant hardware configuration is simply no longer true. The new assumption one should start from when analysing this issue is that when the system boots, we cannot know what devices we will find, and we can not know if the devices we do find are the same ones we had when the system was last shut down.

And in fact, this is very much the case with laptops today: if I attach my IOmega Zip drive to my laptop it appears like a SCSI disk named "/dev/da0", but so does the RAID-5 array attached to the PCI SCSI controller installed in my laptop's docking station. If I change mode to "a+rw" on the Zip drive, do I want that mode to apply to the RAID-5 as well? Unlikely.

And what if we have persistent information about the mode of device "/dev/sio0", but we boot and do not find any sio devices? Do we keep the information in our device-persistence registry? How long do we keep it? If I borrow a a modem card, set the permissions to some non-standard value like 0666, and then attach some other serial device a year from now - do I want some old permissions changes to come back and haunt me, just because they both happened to be "/dev/sio0"? Unlikely.

The fact that more people have laptop computers today than five years ago, and the fact that nobody has been able to credibly propose where a persistent DEVFS would actually store the information about these things in the first place has settled the issue.

Persistence may be the right answer, but to the wrong question: persistence is not a desirable property for a DEVFS when the hardware configuration may change literally at any time.

6.2. Who decides on the names?

In a DEVFS-enabled system, the responsibility for creating nodes in /dev shifts to the device drivers, and consequently the device drivers get to choose the names of the device files. In addition an initial value for owner, group and mode bits are provided by the device driver.

But should it be possible to rename "/dev/lpt0" to "/dev/myprinter"? While the obvious affirmative answer is easy to arrive at, it leaves a lot to be desired once the implications are unmasked.

Most device drivers know their own name and use it purposefully in their debug and log messages to identify themselves. Furthermore, the "NewBus" [NewBus] infrastructure facility, which ties hardware to device drivers, identifies things by name and unit numbers.

A very common way to report errors in fact:

```
#define LPT_NAME "lpt" /* our official name */
[...]
printf(LPT_NAME
    ": cannot alloc ppbus (%d)!", error);
```

So despite the user renaming the device node pointing to the printer to "myprinter", this has absolutely no effect in the kernel and can be considered a userland aliasing operation.

The decision was therefore made that it should not be possible to rename device nodes since it would only lead to confusion and because the desired effect could be attained by giving the user the ability to create symlinks in DEVFS.

6.3. On-demand device creation

Pseudo-devices like pty, tun and bpf, but also some real devices, may not pre-emptively create entries for all possible device nodes. It would be a pointless waste of resources to always create 1000 ptys just in case they are needed, and in the worst case more than 1800 device nodes would be needed per physical disk to represent all possible slices and partitions.

For pseudo-devices the task at hand is to make a magic device node, "/dev/pty", which when opened will magically transmogrify into the first available pty subdevice, maybe "/dev/pty123".

Device submodes, on the other hand, work by having multiple entries in /dev, each with a different minor number, as a way to instruct the device driver in aspects of its operation. The most widespread example is probably "/dev/mt0" and "/dev/nmt0", where the node with the extra "n" instructs the tape device driver to not rewind on close.[5]

Some UNIX systems have solved the problem for pseudo-devices by creating magic cloning devices like "/dev/tcp". When a cloning device is opened, it finds a free instance and through vnode and file descriptor mangling return this new device to the opening process.

This scheme has two disadvantages: the complexity of switching vnodes in midstream is non-trivial, but even worse is the fact that it does not work for submodes for a device because it only reacts to one particular /dev entry.

The solution for both needs is a more flexible on-demand device creation, implemented in FreeBSD as a two-level lookup. When a filename is looked up in

[5] This is the answer to the question in footnote number 2.

DEVFS, a match in the existing device nodes is sought first and if found, returned. If no match is found, device drivers are polled in turn to ask if they would be able to synthesise a device node of the given name.

The device driver gets a chance to modify the name and create a device with make_dev(). If one of the drivers succeeds in this, the lookup is started over and the newly found device node is returned:

```
pty_clone()
   if (name != "pty")
      return(NULL); /* no luck */
   n = find_next_unit();
   dev = make_dev(...,n,"pty%d",n);
   name = dev->name;
   return(dev);
```

An interesting mixed use of this mechanism is with the sound device drivers. Modern sound devices have multiple channels, presumably to allow the user to listen to CNN, Napstered MP3 files and Quake sound effects at the same time. The only problem is that all applications attempt to open "/dev/dsp" since they have no concept of multiple sound devices. The sound device drivers use the cloning facility to direct "/dev/dsp" to the first available sound channel completely transparently to the process.

There are very few drawbacks to this mechanism, the major one being that "ls /dev" now errs on the sparse side instead of the rich when used as a system device inventory, a practice which has always been of dubious precision at best.

6.4. Deleting and recreating devices

Deleting device nodes is no problem to implement, but as likely as not, some people will want a method to get them back. Since only the device driver know how to create a given device, recreation cannot be performed solely on the basis of the parameters provided by a process in userland.

In order to not complicate the code which updates the directory structure for a mountpoint to reflect changes in the DEVFS inode list, a deleted entry is merely marked with DE_WHITEOUT instead of being removed entirely. Otherwise a separate list would be needed for inodes which we had deleted so that they would not be mistaken for new inodes.

The obvious way to recreate deleted devices is to let mknod(2) do it by matching the name and disregarding the major/minor arguments. Recreating the device with mknod(2) will simply remove the DE_WHITEOUT flag.

6.5. Jail(2), chroot(2) and DEVFS

The primary requirement from facilities like jail(2) and chroot(2) is that it must be possible to control the contents of a DEVFS mount point.

Obviously, it would not be desirable for dynamic devices to pop into existence in the carefully pruned /dev of jails so it must be possible to mark a DEVFS mountpoint as "no new devices". And in the same way, the jailed root should not be able to recreate device nodes which the real root has removed.

These behaviours will be controlled with mount options, but these have not yet been implemented because FreeBSD has run out of bitmap flags for mount options, and a new unlimited mount option implementation is still not in place at the time of writing.

One mount option "jaildevfs", will restrict the contents of the DEVFS mountpoint to the "normal set" of devices for a jail and automatically hide all future devices and make it impossible for a jailed root to unhide hidden entries while letting an un-jailed root do so.

Mounting or remounting read-only, will prevent all future devices from appearing and will make it impossible to hide or un-hide entries in the mountpoint. This is probably only useful for chroots or jails where no tty access is intended since cloning will not work either.

More mount options may be needed as more experience is gained.

6.6. Default mode, owner & group

When a device driver creates a device node, and a DEVFS mount adds it to its directory tree, it needs to have some values for the access control fields: mode, owner and group.

Currently, the device driver specifies the initial values in the make_dev() call, but this is far from optimal. For one thing, embedding magic UIDs and GIDs in the kernel is simply bad style unless they are numerically zero. More seriously, they represent compile-time defaults which in these enlightened days is rather old-fashioned.

7. Cleaning up before we build: struct specinfo and dev_t

Most of the rest of the paper will be about the various challenges and issues in the implementation of DEVFS in FreeBSD. All of this should be applicable to other systems derived from 4.4BSD-Lite as well.

POSIX has defined a type called "dev_t" which is the identity of a device. This is mainly for use in the few system calls which knows about devices: stat(2), fstat(2) and mknod(2). A dev_t is constructed by

logically OR'ing the major# and minor# for the device. Since those have been defined as having no overlapping bits, the major# and minor# can be retrieved from the dev_t by a simple masking operation.

Although the kernel had a well-defined concept of any particular device it did not have a data structure to represent "a device". The device driver has such a structure, traditionally called "softc" but the high kernel does not (and should not!) have access to the device driver's private data structures.

It is an interesting tale how things got to be this way,[6] but for now just record for a fact how the actual relationship between the data structures was in the 4.4BSD release (Fig. I). [44BSDBook]

As for all other files, a vnode references a filesystem inode, but in addition it points to a "specinfo" structure. In the inode we find the dev_t which is used to reference the device driver.

Access to the device driver happens by extracting the major# from the dev_t, indexing through the global devsw[] array to locate the device driver's entry point.

The device driver will extract the minor# from the dev_t and use that as the index into the softc array of private data per device.

The "specinfo" structure is a little sidekick vnodes grew underway, and is used to find all vnodes which reference the same device (i.e. they have the same major# and minor#). This linkage is used to determine which vnode is the "chosen one" for this device, and to keep track of open(2)/close(2) against this device. The actual implementation was an inefficient hash implementation, which depending on the vnode reclamation rate and /dev directory lookup traffic, may become a measurable performance liability.

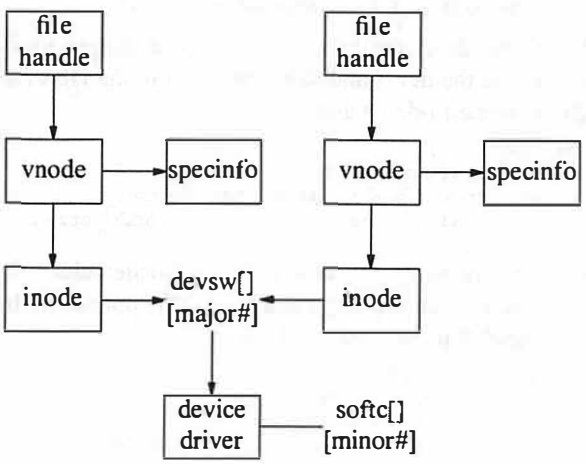

Fig. 1 - Data structures in 4.4BSD

[6] Basically, devices should have been moved up with sockets and pipes at the file descriptor level when the VFS layering was introduced, rather than have all the special casing throughout the vnode system.

7.1. The new vnode/inode/dev_t layout

In the new layout (Fig. 2) the specinfo structure takes a central role. There is only one instanace of struct specinfo per device (i.e. unique major# and minor# combination) and all vnodes referencing this device point to this structure directly.

In userland, a dev_t is still the logical OR of the major# and minor#, but this entity is now called a udev_t in the kernel. In the kernel a dev_t is now a pointer to a struct specinfo.

All vnodes referencing a device are linked to a list hanging directly off the specinfo structure, removing the need for the hash table and consequently simplifying and speeding up a lot of code dealing with vnode instantiation, retirement and name-caching.

The entry points to the device driver are stored in the specinfo structure, removing the need for the devsw[] array and allowing device drivers to use separate entry-points for various minor numbers.

This is is very convenient for devices which have a "control" device for management and tuning. The control device, almost always have entirely separate open/close/ioctl implementations [MD.C].

In addition to this, two data elements are included in the specinfo structure but "owned" by the device driver. Typically the device driver will store a pointer

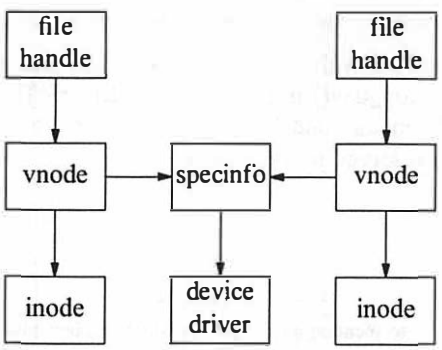

Fig. 2 - The new FreeBSD data structures.

to the softc structure in one of these, and unit number or mode information in the other.

This removes the need for drivers to find the softc using array indexing based on the minor#, and at the same time has obliviated the need for the compiled-in "NFOO" constants which traditionally determined how many softc structures and therefore devices the driver could support.[7]

There are some trivial technical issues relating to allocating the storage for specinfo early in the boot sequence and how to find a specinfo from the udev_t/major#+minor#, but they will not be discussed here.

7.2. Creating and destroying devices

Ideally, devices should only be created and destroyed by the device drivers which know what devices are present. This is accomplished with the make_dev() and destroy_dev() function calls.

Life is seldom quite that simple. The operating system might be called on to act as a NFS server for a diskless workstation, possibly even of a different architecture, so we still need to be able to represent device nodes with no device driver backing in the filesystems and consequently we need to be able to create a specinfo from the major#+minor# in these inodes when we encounter them. In practice this is quite trivial, but in a few places in the code one needs to be aware of the existence of both "named" and "anonymous" specinfo structures.

The make_dev() call creates a specinfo structure and populates it with driver entry points, major#, minor#, device node name (for instance "lpt0"), UID, GID and access mode bits. The return value is a dev_t (i.e., a pointer to struct specinfo). If the device driver determines that the device is no longer present, it calls destroy_dev(), giving a dev_t as argument and the dev_t will be cleaned and converted to an anonymous dev_t.

Once created with make_dev() a named dev_t exists until destroy_dev() is called by the driver. The driver can rely on this and keep state in the fields in dev_t which is reserved for driver use.

[7] Not to mention all the drivers which implemented panic(2) because they forgot to perform bounds checking on the index before using it on their softc arrays.

8. DEVFS

By now we have all the relevant information about each device node collected in struct specinfo but we still have one problem to solve before we can add the DEVFS filesystem on top of it.

8.1. The interrupt problem

Some device drivers, notably the CAM/SCSI subsystem in FreeBSD will discover changes in the device configuration inside an interrupt routine.

This imposes some limitations on what can and should do be done: first one should minimise the amount of work done in an interrupt routine for performance reasons; second, to avoid deadlocks, vnodes and mountpoints should not be accessed from an interrupt routine.

Also, in addition to the locking issue, a machine can have many instances of DEVFS mounted: for a jail(8) based virtual-machine web-server several hundred instances is not unheard of, making it far too expensive to update all of them in an interrupt routine.

The solution to this problem is to do all the filesystem work on the filesystem side of DEVFS and use atomically manipulated integer indices ("inode numbers") as the barrier between the two sides.

The functions called from the device drivers, make_dev(), destroy_dev() &c. only manipulate the DEVFS inode number of the dev_t in question and do not even get near any mountpoints or vnodes.

For make_dev() the task is to assign a unique inode number to the dev_t and store the dev_t in the DEVFS-global inode-to-dev_t array.

```
make_dev(...)
    store argument values in dev_t
    assign unique inode number to dev_t
    atomically insert dev_t into inode_array
```

For destroy_dev() the task is the opposite: clear the inode number in the dev_t and NULL the pointer in the devfs-global inode-to-dev_t array.

```
destroy_dev(...)
    clear fields in dev_t
    zero dev_t inode number.
    atomically clear entry in inode_array
```

Both functions conclude by atomically incrementing a global variable `devfs_generation` to leave an indication to the filesystem side that something has changed.

By modifying the global state only with atomic instructions, locks have been entirely avoided in this part of the code which means that the make_dev() and destroy_dev() functions can be called from practically anywhere in the kernel at any time.

On the filesystem side of DEVFS, the only two vnode methods which examine or rely on the directory structure, VOP_LOOKUP and VOP_READDIR, call the function devfs_populate() to update their mountpoint's view of the device hierarchy to match current reality before doing any work.

```
devfs_readdir(...)
    devfs_populate(...)
    ...
```

The devfs_populate() function, compares the current devfs_generation to the value saved in the mountpoint last time devfs_populate() completed and if (actually: while) they differ a linear run is made through the devfs-global inode-array and the directory tree of the mountpoint is brought up to date.

The actual code is slightly more complicated than shown in the pseudo-code here because it has to deal with subdirectories and hidden entries.

```
devfs_populate(...)
  while (mount->generation != devfs_generation)
    for i in all inodes
      if inode created)
        create directory entry
      else if inode destroyed
        remove directory entry
```

Access to the global DEVFS inode table is again implemented with atomic instructions and failsafe retries to avoid the need for locking.

From a performance point of view this scheme also means that a particular DEVFS mountpoint is not updated until it needs to be, and then always by a process belonging to the jail in question thus minimising and distributing the CPU load.

9. Device-driver impact

All these changes have had a significant impact on how device drivers interact with the rest of the kernel regarding registration of devices.

If we look first at the "before" image in Fig. 3, we notice first the NFOO define which imposes a firm upper limit on the number of devices the kernel can deal with. Also notice that the softc structure for all of them is allocated at compile time. This is because most device drivers (and texts on writing device drivers) are from before the general kernel malloc facility [Mckusick1988] was introduced into the BSD kernel.

```
#ifndef NFOO
#       define NFOO    4
#endif

struct foo_softc {
        ...
} foo_softc[NFOO];

int nfoo = 0;

foo_open(dev, ...)
{
        int unit = minor(dev);
        struct foo_softc *sc;

        if (unit >= NFOO || unit >= nfoo)
                return (ENXIO);

        sc = &foo_softc[unit]

        ...
}

foo_attach(...)
{
        struct foo_softc *sc;
        static int once;

        ...
        if (nfoo >= NFOO) {
                /* Have hardware, can't handle */
                return (-1);
        }
        sc = &foo_softc[nfoo++];
        if (!once) {
                cdevsw_add(&cdevsw);
                once++;
        }
        ...
}
```

Fig. 3 - Device-driver, old style.

Also notice how range checking is needed to make sure that the minor# is inside range. This code gets more complex if device-numbering is sparse. Code equivalent to that shown in the foo_open() routine would also be needed in foo_read(), foo_write(), foo_ioctl() &c.

Finally notice how the attach routine needs to remember to register the cdevsw structure (not shown) when the first device is found.

Now, compare this to our "after" image in Fig. 4. NFOO is totally gone and so is the compile time allocation of space for softc structures.

The foo_open (and foo_close, foo_ioctl &c) functions can now derive the softc pointer directly from the dev_t they receive as an argument.

```
struct foo_softc {
        ....
};

int nfoo;

foo_open(dev, ...)
{
        struct foo_softc *sc = dev->si_drv1;

        ...
}

foo_attach(...)
{
        struct foo_softc *sc;

        ...
        sc = MALLOC(..., M_ZERO);
        if (sc == NULL) {
                /* Have hardware, can't handle */
                return (-1);
        }
        sc->dev = make_dev(&cdevsw, nfoo,
            UID_ROOT, GID_WHEEL, 0644,
            "foo%d", nfoo);
        nfoo++;
        sc->dev->si_drv1 = sc;
        ...
}
```

Fig. 4 - Device-driver, new style.

In foo_attach() we can now attach to all the devices we can allocate memory for and we register the cdevsw structure per dev_t rather than globally.

This last trick is what allows us to discard all bounds checking in the foo_open() &c. routines, because they can only be called through the cdevsw, and the cdevsw is only attached to dev_t's which foo_attach() has created. There is no way to end up in foo_open() with a dev_t not created by foo_attach().

In the two examples here, the difference is only 10 lines of source code, primarily because only one of the worker functions of the device driver is shown. In real device drivers it is not uncommon to save 50 or more lines of source code which typically is about a percent or two of the entire driver.

10. Future work

Apart from some minor issues to be cleaned up, DEVFS is now a reality and future work therefore is likely concentrate on applying the facilities and functionality of DEVFS to FreeBSD.

10.1. devd

It would be logical to complement DEVFS with a "device-daemon" which could configure and de-configure devices as they come and go. When a disk appears, mount it. When a network interface appears, configure it. And in some configurable way allow the user to customise the action, so that for instance images will automatically be copied off the flash-based media from a camera, &c.

In this context it is good to question how we view dynamic devices. If for instance a printer is removed in the middle of a print job and another printer arrives a moment later, should the system automatically continue the print job on this new printer? When a disk-like device arrives, should we always mount it? Should we have a database of known disk-like devices to tell us where to mount it, what permissions to give the mount-point? Some computers come in multiple configurations, for instance laptops with and without their docking station. How do we want to present this to the users and what behaviour do the users expect?

10.2. Pathname length limitations

In order to simplify memory management in the early stages of boot, the pathname relative to the mountpoint is presently stored in a small fixed size buffer inside struct specinfo. It should be possible to use filenames as long as the system otherwise permits, so some kind of extension mechanism is called for.

Since it cannot be guaranteed that memory can be allocated in all the possible scenarios where make_dev() can be called, it may be necessary to mandate that the caller allocates the buffer if the content will not fit inside the default buffer size.

10.3. Initial access parameter selection

As it is now, device drivers propose the initial mode, owner and group for the device nodes, but it would be more flexible if it were possible to give the kernel a set of rules, much like packet filtering rules, which allow the user to set the wanted policy for new devices. Such a mechanism could also be used to filter new devices for mount points in jails and to determine other behaviour.

Doing these things from userland results in some awkward race conditions and software bloat for embedded systems, so a kernel approach may be more suitable.

10.4. Applications of on-demand device creation

The facility for on-demand creation of devices has some very interesting possibilities.

One planned use is to enable user-controlled labelling of disks. Today disks have names like /dev/da0,

/dev/ad4, but since this numbering is topological any change in the hardware configuration may rename the disks, causing /etc/fstab and backup procedures to get out of sync with the hardware.

The current idea is to store on the media of the disk a user-chosen disk name and allow access through this name, so that for instance /dev/mydisk0 would be a symlink to whatever topological name the disk might have at any given time.

To simplify this and to avoid a forest of symlinks, it will probably be decided to move all the sub-divisions of a disk into one subdirectory per disk so just a single symlink can do the job. In practice that means that the current /dev/ad0s2f will become something like /dev/ad0/s2f and so on. Obviously, in the same way, disks could also be accessed by their topological address, down to the specific path in a SAN environment.

Another potential use could be for automated offline data media libraries. It would be quite trivial to make it possible to access all the media in the library using /dev/lib/$LABEL which would be a remarkable simplification compared with most current automated retrieval facilities.

Another use could be to access devices by parameter rather than by name. One could imagine sending a printjob to /dev/printer/color/A2 and behind the scenes a search would be made for a device with the correct properties and paper-handling facilities.

11. Conclusion

DEVFS has been successfully implemented in FreeBSD, including a powerful, simple and flexible solution supporting pseudo-devices and on-demand device node creation.

Contrary to the trend, the implementation added functionality with a net decrease in source lines, primarily because of the improved API seen from device drivers point of view.

Even if DEVFS is not desired, other 4.4BSD derived UNIX variants would probably benefit from adopting the dev_t/specinfo related cleanup.

12. Acknowledgements

I first got started on DEVFS in 1989 because the abysmal performance of the Olivetti M250 computer forced me to implement a network-disk-device for Minix in order to retain my sanity. That initial work led to a crude but working DEVFS for Minix, so obviously both Andrew Tannenbaum and Olivetti deserve credit for inspiration.

Julian Elischer implemented a DEVFS for FreeBSD around 1994 which never quite made it to maturity and subsequently was abandoned.

Bruce Evans deserves special credit not only for his keen eye for detail, and his competent criticism but also for his enthusiastic resistance to the very concept of DEVFS.

Many thanks to the people who took time to help me stamp out "Danglish" through their reviews and comments: Chris Demetriou, Paul Richards, Brian Somers, Nik Clayton, and Hanne Munkholm. Any remaining insults to proper use of english language are my own fault.

13. References

[44BSDBook] Mckusick, Bostic, Karels & Quarterman: "The Design and Implementation of 4.4 BSD Operating System." Addison Wesley, 1996, ISBN 0-201-54979-4.

[Heidemann91a] John S. Heidemann: "Stackable layers: an architecture for filesystem development." Master's thesis, University of California, Los Angeles, July 1991. Available as UCLA technical report CSD-910056.

[Kamp2000] Poul-Henning Kamp and Robert N. M. Watson: "Confining the Omnipotent root." Proceedings of the SANE 2000 Conference. Available in FreeBSD distributions in `/usr/share/papers`.

[MD.C] Poul-Henning Kamp et al: FreeBSD memory disk driver: `src/sys/dev/md/md.c`

[Mckusick1988] Marshall Kirk Mckusick, Mike J. Karels: "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX-Kernel" Proceedings of the San Francisco USENIX Conference, pp. 295-303, June 1988.

[Mckusick1999] Dr. Marshall Kirk Mckusick: Private email communication. *"According to the SCCS logs, the chroot call was added by Bill Joy on March 18, 1982 approximately 1.5 years before 4.2BSD was released. That was well before we had ftp servers of any sort (ftp did not show up in the source tree until January 1983). My best guess as to its purpose was to allow Bill to chroot into the /4.2BSD build directory and build a system using only the files, include files, etc contained in that tree. That was the only use of chroot that I remember from the early days."*

[Mckusick2000] Dr. Marshall Kirk Mckusick: Private communication at BSDcon2000 conference. *"I have not used block devices since I wrote FFS and that was*

many *years ago.*"

[NewBus] NewBus is a subsystem which provides most of the glue between hardware and device drivers. Despite the importance of this there has never been published any good overview documentation for it. The following article by Alexander Langer in "Dæmonnews" is the best reference I can come up with: `http://www.daemonnews.org/200007/newbus-intro.html`

[Pike2000] Rob Pike: "Systems Software Research is Irrelevant."
`http://www.cs.bell-labs.com/who/rob/utah2000.pdf`

[Pike90a] Rob Pike, Dave Presotto, Ken Thompson and Howard Trickey: "Plan 9 from Bell Labs." Proceedings of the Summer 1990 UKUUG Conference.

[Pike92a] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey and Phil Winterbottom: "The Use of Name Spaces in Plan 9." Proceedings of the 5th ACM SIGOPS Workshop.

[Raspe1785] Rudolf Erich Raspe: "Baron Münchhausen's Narrative of his marvellous Travels and Campaigns in Russia." Kearsley, 1785.

[Ritchie74] D.M. Ritchie and K. Thompson: "The UNIX Time-Sharing System" Communications of the ACM, Vol. 17, No. 7, July 1974.

[Ritchie98] Dennis Ritchie: private conversation at USENIX Annual Technical Conference New Orleans, 1998.

[Thompson78] Ken Thompson: "UNIX Implementation" The Bell System Technical Journal, vol 57, 1978, number 6 (part 2) p. 1931ff.

Resisting SYN flood DoS attacks with a SYN cache

Jonathan Lemon jlemon@FreeBSD.org
FreeBSD Project

Abstract

Machines that provide TCP services are often susceptible to various types of Denial of Service attacks from external hosts on the network. One particular type of attack is known as a SYN flood, where external hosts attempt to overwhelm the server machine by sending a constant stream of TCP connection requests, forcing the server to allocate resources for each new connection until all resources are exhausted. This paper discusses several approaches for dealing with the exhaustion problem, including SYN caches and SYN cookies. The advantages and drawbacks of each approach are presented, and the implementation of the specific solution used in FreeBSD is analyzed.

1 Introduction

The Internet today is driven by machines that communicate using services layered on top of the TCP/IP protocols, these include HTTP, ftp and ssh, among others. The accessibility of these services is dependent on how well the underlying transport protocol performs, which in this case is TCP. If TCP is unable or unavailable to deliver the layered service to a remote machine, the user perceives the site as being dead or inaccessible. Perhaps merely an inconvenience in the past, this is a much more serious problem today as machines are being used for commerce and business.

One way that a malicious host can attempt to deny services provided by a server machine is by sending a large number of TCP open requests. These are known as SYN packets, named after the specific bit in the TCP specification, hence this type of Denial of Service attack is often called SYN bombing or SYN flooding. When the server receives this packet, it is interpreted as a request by the remote host to initiate a TCP connection, and at this point, the OS on the server machine commonly allocates resources to track the TCP state. By sending these requests in rapid succession, an attacker can exhaust the resources on the machine to the point where it becomes unresponsive, or crashes.

The server can attempt to reduce the impact of the flooding by changing the resource allocation strategy that it uses. One approach is to allocate minimal state when the initial request is received, and only allocate all the resources required when the connection is completed; this is termed a SYN cache. Another approach is to allocate no state, but instead send a cryptographic secret back to the originator, called a cookie; hence the name SYN cookie. Both approaches are intended to allow the machine to continue to provide its services to valid users.

The rest of this paper is structured as follows: Section 2 examines the details involved in the SYN flood Denial of Service attacks and examines the approaches of different defenses. Section 3 details the experimental setup used for testing, while Section 4 describes the current system behavior and motivation for change. Section 5 discusses the SYN cache implementation and presents the performance measurements from the change, while Section 6 does the same for SYN cookies. Section 7 discusses related work, and the paper concludes with a summary in Section 8.

2 TCP Denial of Service

A traditional TCP 3 way handshake for establishing connections is shown in Figure 1, where state is allocated on the server side upon receipt of the SYN to hold information associated with the incomplete connection. The goal of a SYN flood is to tie up resources on the server machine, so that it is unable to respond to legitimate connections. This is accomplished by having the client discard the returning SYN,ACK from the server and not send the final ACK. This results in the server retaining the partial state that was allocated from the initial SYN.

The attacker does not necessarily have to be on a fast machine or network to accomplish this. Standard TCP

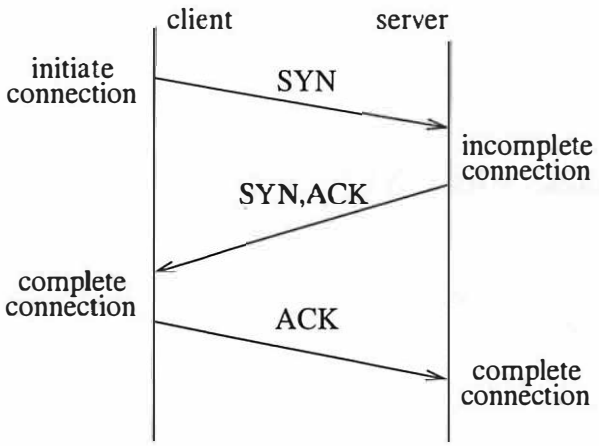

Figure 1: Standard TCP 3 way handshake.

will not time out connections until a certain number of retransmits have been made, which usually is a total of 511 seconds[7]. Assuming a machine permits a maximum of 1024 incomplete connections per socket, this means an attacker has only to send 2 connection attempts per second to exhaust all allocated resources. In practice, this does not form a DoS attack, as existing incomplete connections are dropped when a new SYN request is received. The time required for the server to send a SYN,ACK and have the client reply is known as the round trip time (RTT); if an ACK arrives at the server but does not find a corresponding incomplete connection state, the server will not establish a connection. By forcing the server to drop incomplete connection state at a rate larger than the RTT, an attacker is able to insure that no connections are able to complete.

Each connection is dropped with $1/N$ probability, and if the goal is to recycle every connection before the average RTT, an attacker would need to flood the machine at a rate of N/RTT packets per second. For a listen queue size of 1024, and a 100 millisecond RTT, this results in about 10,000 packets per second. A minimal size TCP packet is 64 bytes, so the total bandwidth used is only 4Mb/second, within the realm of practicality.

As the sender may forge their source IP address, a defense that relies on filtering packets based on the source IP will not be effective in all cases. Using a random source IP address will also cause more resources to be tied up on the server if a per-IP route structure is allocated.

Often it is not possible to distinguish attacks from real connection attempts, other than by observing the volume of SYNs that are arriving at the server, so the machine needs to be able to handle them in some fashion.

In order to defend against this type of attack, the amount of the amount of state that is allocated should be reduced, or even eliminated completely by delaying allocation until the connection is completed. Two known approaches to do this are known as SYN cache and SYN cookies. The caching approach is similar to the existing behavior, but allocates a much smaller state structure to record the initial connection request, while the cookie approach attempts to encode the state in a small quantity which is returned by the client when the TCP handshake is completed.

2.1 Defenses

SYN caching allocates some state on the machine, but even with this reduced state it is possible to encounter resource exhaustion. The code must be prepared to handle state overflows and choose which items to drop in order to preserve fairness.

The initial SYN request carries a collection of options which apply the TCP connection, these commonly include the desired MSS, requested window scaling for the connection, use of timestamps, and various other items. Part of the purpose of the allocated state is to record these options, which are not retransmitted in the return ACK from the client.

SYN cookies do not store any state on the machine, but keep all state regarding the initial TCP connection in the network, treating it as an infinitely deep queue. This is done by use of a cryptographic function to encode all information into a value that is sent to the client with the SYN,ACK and returned to the server in the final portion of the 3 way handshake. While this approach appears attractive, it has the drawback of not being able to encode all the TCP options from the initial SYN into the cookie. These options are then lost, denying the use of certain TCP performance enhancements.

A secondary problem related to cookies is that the TCP protocol requires unacknowledged data to be retransmitted. The server is supposed to retransmit the SYN,ACK before giving up and dropping the connection, whereupon a RST is sent to the client in order to shutdown the connection. When SYN,ACK arrives at a client but the return ACK is lost, this results in a disparity about the established state between the client and server. Ordinarily, this case will be handled by server retransmits, but in the case of SYN cookies, there is no state kept on the server, and a retransmission is not possible.

SYN cookies also have the property that the entire connection establishment is performed by the returning ACK, independent of the preceding SYN and SYN,ACK transmissions. This opens the possibility of flooding the server with ACK requests, in hopes that one will contain the correct value which allows a connection to be established. This also provides an approach to bypass firewalls which restrict external connections by filtering

out incoming packets which have the SYN bit set, since initial SYN packet is no longer required to establish a connection.

Another difficulty with cookies is that they are incompatible with transactional TCP[6]. T/TCP works by sending monotonically increasing sequence numbers to the peer in the TCP options field, and uses previously received sequence numbers to establish connections on the initial SYN, eliminating the 3 way handshake. However, use of the T/TCP sequence numbers is mandatory once a TCP connection is initiated, and this requires the server to record the initial sequence number, and whether the T/TCP option was requested.

Thus cookies cannot be used as the normal line of defense in a high performance server. The usual approach is to use a state allocation mechanism, and fall back to using cookies only after a certain amount of state has been allocated. This is the approach taken by the the Linux kernel implementation.

3 Experimental Setup

The code base used was FreeBSD 4.4-stable, from sources as of November 14th, 2001. The target machine used for testing was an Intel PIII/850, with 320MB of memory, and was equipped with an onboard Intel EtherExpress 100Mb/s chip, an Intel 1000/Pro Gigabit adapter and a NetGear GA620 Gigabit adapter. The NetGear adapter was attached directly to a second machine that acted as a packet source, while the Intel adapter was directly attached to a third machine that acted as a packet sink. A fourth machine was connected via the 100Mb port and was used for taking timing measurements of real connection requests to the test machine.

A default route was installed on the test machine so that all incoming traffic from the source was sent out to the sink via the other gigabit link. The kern.ipc.somaxconn parameter, which controls the maximum listen backlog, was raised to 1024, while net.inet.tcp.msl was turned down to 30 milliseconds in order not to run out of TCP ports. Mbufs and mbuf clusters were set to 65536 and 16384 respectively, and the system was monitored to insure that the mbuf limit was not reached.

When SYN flooding the box, the source was configured to generate SYN packets at a rate of 15,000 packets per second. This rate was chosen as a load that the box could reasonably handle without becoming susceptible to receiver livelock. Under this load, the box was handling upwards of 30,000 packets per second, incoming and outgoing. The source addresses of the SYN packets were randomly chosen from the 10.x.x.x subnet, and the source port numbers and ISS were also randomly generated.

A small program that accepted and closed incoming connections was run on the test machine, in order to provide a listen socket for incoming packets. Timing measurements were taken on the control machine that was attached to the 100Mb port, which involved taking 2000 samples of the amount of time required for a connect() call to complete to the target machine.

4 Motivation

Initial tests were performed on the target machine using an unmodified 4.4-stable kernel while undergoing SYN flooding. The size of the listen socket backlog was varied from the default 128 entries to 1024 entries, as permitted by kern.ipc.somaxconn. The results of the test are presented in Figure 2.

In this test, with a backlog of 128 connections, 90% of the 2000 connections initiated to the target machine complete within 500ms. When the application specifies a backlog of 1024 connections in the listen() call, only 2.5% of the connections complete within the same time period.

The dropoff in performance here may be attributed to the fact that the sodropablereq() function does not scale. The goal of this function is to provide a random drop of incomplete connections from the listen queue, in order to insure fairness.

However, the queue is kept on a linear list, and in order to drop a random element, a list traversal is required to reach the target element. This means that on average, 1/2 of the total length of the queue must be traversed to reach the element; for a listen queue backlog of 1024 elements, this leads to an average of $(3*(1024/2))/2$, or 768 elements traversed for each incoming SYN.

Profiling results show that in this particular case, the system spends 30% of its time in sodropablereq(), and subjectively, is almost completely unresponsive. Examining the graph, we see that there is a considerable dropoff in performance between the backlog cases of 768 entries and 1024 entries, the reason of which is unclear. It is likely that there is a 'knee' in the performance curve is between these points, and system may have reached a point of saturation.

For the rest of the paper, a listen queue backlog of 1024 entries is used, as this is a realistic value used on production systems[4]. It also serves to illustrate the performance gains from a syncache or syncookie implementation.

4.1 Implementation

The new implementation for FreeBSD provides a SYN cache as the first approach for handling incoming connections, and has every connection pass through the

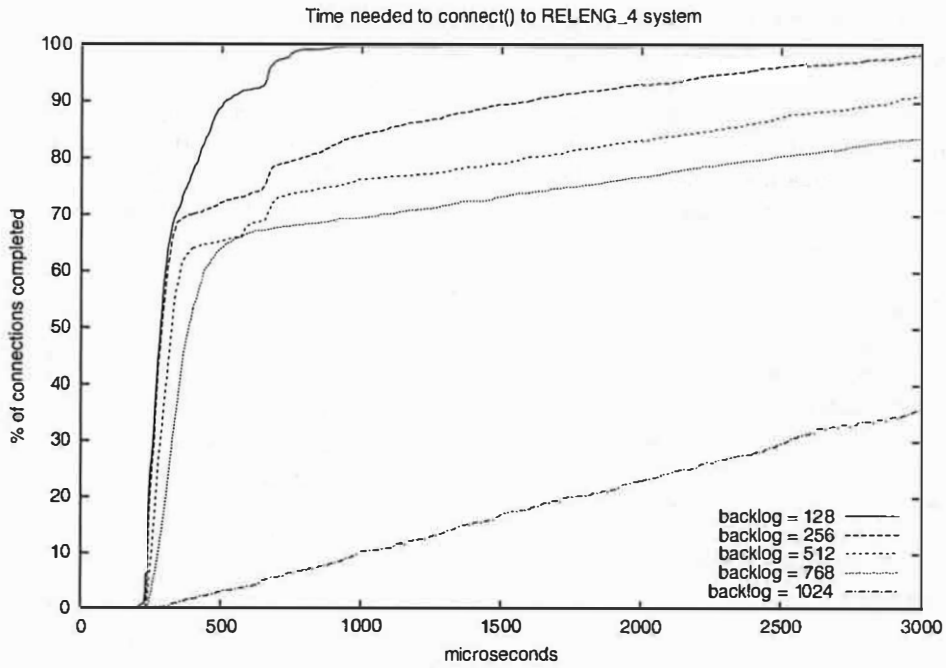

Figure 2: Time needed to connect() to a RELENG_4 system under a SYN flood attack. The kern.ipc.somaxconn parameter on the remote machine was set to 1024, and the size of the listen backlog was varied for each run.

cache. If an existing entry in the cache needs to be evicted, a sysctl tunable controls the optional behavior of sending back a SYN cookie instead of evicting the entry from the cache. In the following discussion, first the implementation of the syncache will be presented, independent of syncookies, with the next section explaining how syncookies modify the behavior of the syncache.

5 SYN Cache

The syncache implementation replaces the per-socket linear chain of incomplete queued connections with a global hashtable, which provides two forms of protection against running out of resources. These are a limit on the the total number of entries in the table, which provides an upper bound on the amount of memory that the syncache takes up, and a limit on the number of entries in a given hash bucket. The latter limit bounds the amount of time that the machine needs to spend searching for a matching entry, as well as limiting replacement of the cache entries to a subset of the entire cache. A global table was chosen instead of a per-socket table as it was felt this would be a more efficient use of system resources. A current implementation restriction that all kernel virtual address space for the memory used at interrupt time must be pre-allocated was also a factor in this decision.

One of the major bottlenecks in the original code was the random drop implementation from the linear list, which did not scale. This bottleneck avoided in the syncache, since the queue is split among hash buckets, which are then treated as FIFO queues instead of using random drop. Another way of viewing this is to consider the original linear list partitioned up into a number of sublists equivalent to the size of the hash table, where choosing a bucket enables us to choose which section of the list to drop. Since the hash distribution across the buckets should be uniform, this is an approximate model of choosing a random list entry to drop.

The hash value is computed on the incoming packet using the source and destination addresses, the source and destination port, and a randomly chosen secret. This value is then used as an index into a hash table, where syncache entries are kept on a linked list in each bucket. The secret is used to perturb the hash value so that an attacker cannot target a specific hash bucket and deny service to a specific machine.

While on the surface it may appear that an attacker could implement a DoS by targeting a hash bucket so that a legitimate connection does not reside on the queue long enough to establish a connection, the risks are migitated by the use of the hash secret. Additionally, since the port number of the connecting machine is used in the hash calculations, a second connection attempt from the client machine tends to result in a second hash bucket chosen, further styming any attempt by an attacker to target a specific bucket.

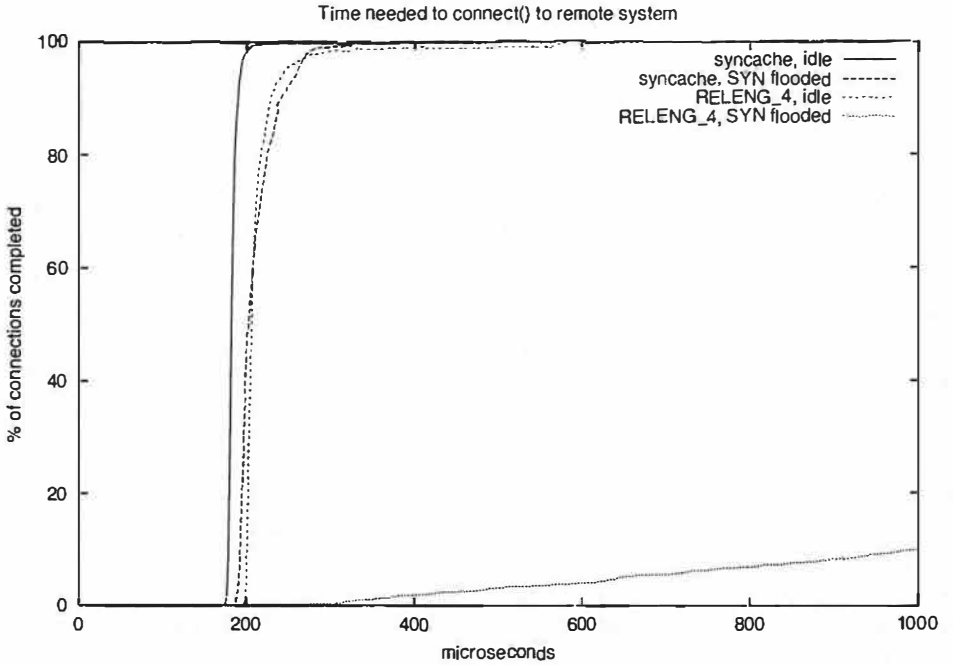

Figure 3: Time needed to connect() to remote system.

If the entry is not found in the bucket, a new syncache entry is created and added to the cache. If the new entry would overflow the per-bucket limit, the oldest entry within that bucket is dropped. If the total number of entries in the cache is exceeded, the oldest entry in the cache is dropped. This way, both the memory usage of the syncache and the amount of CPU time needed to search the hash table are bounded. The user is able to control the sizing of these limits via the following loader tunables established at boot time:

```
net.inet.tcp.syncache.hashsize
net.inet.tcp.syncache.cachelimit
net.inet.tcp.syncache.bucketlimit
```

The *cachelimit* setting determines the maximum number of syncache entries that may be allocated, and bounds the overall memory usage of the system. *hashsize* controls the size of the hash table and should be a power of 2. Finally, *bucketlimit* caps the size of each hash chain, and limits the number of entries that must be searched when looking for a matching SYN entry. However, as the list is handled in FIFO order, an entry must stay on the list for at least one round trip time (RTT) to the remote system in order to successfully establish a connection, so this must be considered when choosing a value for *bucketlimit*.

There are two additional sysctl parameters of interest:

```
net.inet.tcp.syncache.count
net.inet.tcp.syncache.rexmtlimit
```

The first entry is read-only, and indicates how many entries are currently present in the syncache. The second determines how many times a SYN,ACK should be retransmitted to the remote system, and defaults to 3. Three retransmits corresponds to $1 + 2 + 4 + 8 = 15$ seconds, and the odds are that if a connection cannot be established by then, the user has given up.

5.1 Syncache performance

The syncache tests were performed on the target machine using the following system default values: hashsize = 512, cachelimit = 15359, bucketlimit = 30. The results of the test are presented in Figure 3.

As the graph shows, the syncache is effective at handling a SYN flood while still allowing incoming connections. Here, 99% of the incoming connections are completed within 300 microseconds, which is on par with the time required to connect to an idle unmodified system. For comparison, the performance of an unmodified system experiencing a SYN flood is also shown. All of the trials in the test were performed with a listen queue length of 1024.

One interesting result is that the connection latency decreases even when the target box is not experiencing SYN flooding. This is shown by comparing the 'syncache idle' and 'RELENG_4 idle' lines on the graph, which indicate how long it takes to connect to a quiescent system. This result may be attributed to the smaller

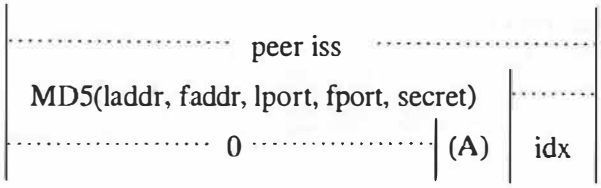

Figure 5: Layers of data in the syncookie.

data structure used to hold the syncache, as the size of the TCP and socket structures allocated and initialized on an unmodified system total 736 bytes, while the smaller syncache structure is only 160 bytes.

6 SYN Cookies

When a syncache bucket does overflow, a fallback mechanism exists which permits sending back a SYN cookie instead of performing oldest FIFO drop of an entry on the hash list. This section explains the syncookie approach, and outlines how the cookie is constructed.

The cookie is sent to the remote system as the system's Initial Sequence Number (ISN), and then returned in the final phase of TCP's three way handshake. As connection establishment is performed by the returning ACK, a secret should be used to validate the connection, which is concealed from the remote system by use of a non-invertible hash. To prevent an intermediate system from collecting cookies and replaying them at a later date, the cookie should also contain a time component. The solution chosen here was to keep a table of secrets which have a bounded lifetime, which has an added benefit of regularly changing the secret which is sent back to the remote system. Figure 5 shows the internal structure of the cookie.

The basis of the implementation is a table of 128 32-bit values obtained from arc4random(). Each entry is used for a duration of 31.25 milliseconds, and has a total lifetime of 4 seconds, which was chosen as a reasonable upper bound for the RTT to the remote system, as SYN,ACK containing the cookie must reach the system and be returned before the secret expires.

In order to generate a cookie, the system tick timer is scaled into units of 31.25 milliseconds by use of divide and shift operations, with the result used to choose the correct window index. If the secret in the current window has expired, a new 32-bit secret is obtained from arc4random(), and the timeout is reset.

The local address, foreign address, local port, foreign port and secret are passed through MD5 to create the initial basis of the cryptographic hash, with 25 bits being used in the cookie, and 7 bits containing the window index. The peer MSS from the TCP options section of the initial SYN is fit into one of 4 predefined MSS values, and the resulting 2 bit index is xor'ed into the mix, as shown by (*A*) in Figure 5. Finally, the peer's 32-bit ISS is xor'ed in to generate the final cookie, which is sent back to the connecting system as the ISN.

Since no state is kept on the server machine, any returning ACK which contains the correct TCP sequence numbers may serve to establish a connection. Validating the ACK is the reverse of the above process. First the peer's sequence number is removed, and then the 7 bit index is used to select the correct window. If the secret has expired, then the ACK is immediately discarded without further processing. This insures that the system does not have to check every incoming ACK unless a syncookie was recently sent. If the timeout indicates that the secret is valid, it is used in the MD5 hash computation. The ACK is considered valid if the remaining 23 bits evaluate to 0.

In practice, this means that a remote system has 4 seconds to try and brute force a space of 2^{23} entries.

6.1 SYN cookie performance

The syncache tests were performed on the target machine by enabling the following sysctl

```
net.inet.tcp.syncookies
```

and then performing the tests in the usual fashion. The results of the test are presented in Figure 4.

The results show that syncookies provides slightly better performance than syncache alone. This may be due to the fact that the syncache calls arc4random() for every SYN,ACK it sends, while the syncookie routines primarily call MD5(). Investigation into the reason for the performance disparity is ongoing, but the results are not available at this time.

There are also a few unusual results here: There does not appear to be a straightforward explanation for the jump in completed connections at 700 microseconds. This is not due to TCP retransmissions, as the first retransmission timeout is set at one second. A possible explanation is that the system is busy executing the interrupt handler for either of the Gigabit adapters, and is delayed in servicing the 100Mb adapter.

Also of interest to note is that while 100% of the syncache connections have completed in 1 second, the same isn't true for syncookies. This shouldn't happen, as no packet loss on the 100Mb segment was observed, and the system did not run out of mbufs. Upon further investigation, this turned out to be a minor bug in the VM system where the initial boot-time allocation request was rounded improperly, leading to a shortage of syncache structure entries. With the current code, at least one entry is always needed in order to send the SYN,ACK reply.

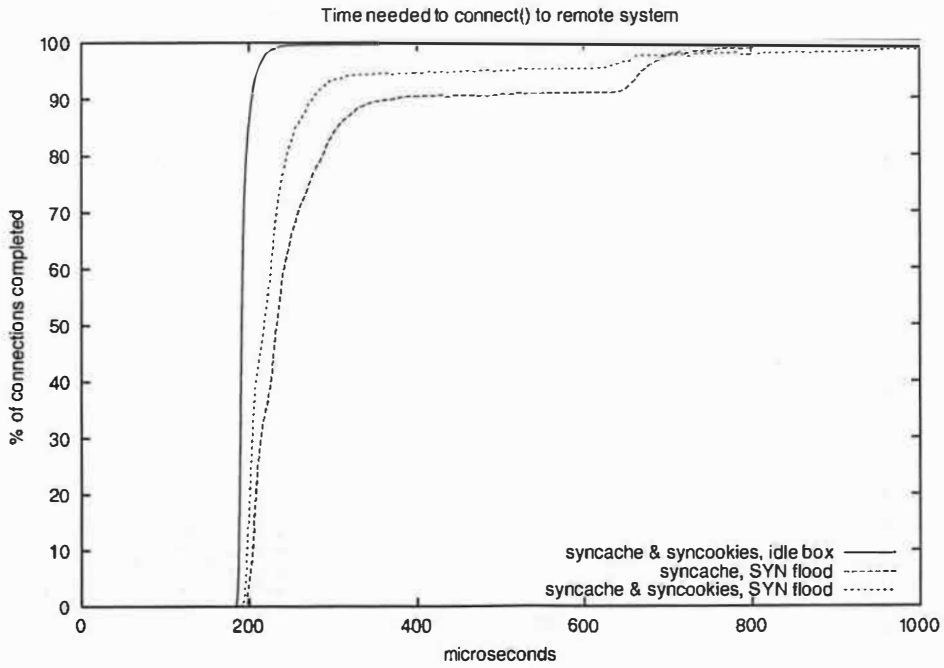

Figure 4: Performance comparison of a system with syncache and syncookies over one using only syncache.

6.2 Round trip performance

Prior measurements were taken by timing how long it takes for a connect() call to complete on the client machine. This corresponds to the time required to complete 2 stages of a TCP handshake, since the client machine enters the ESTABLISHED state as soon as it receives a SYN,ACK. An unanswered question is how long it takes the server to enter the ESTABLISHED state, from the time the initial SYN is sent from the client. This time may be affected by the different processing requirements to verify the ACK, and may fail if the original syncache record no longer exists.

To verify failure was not a concern, the experimental setup was modified to include the time required to read() a byte from the server, which can be viewed as a 4 way handshake: transmit SYN, receive SYN,ACK, transmit ACK, receive data. The results for this test are presented in Figure 6.

On an unloaded box, there is no measurable difference in performance between the syncache and syncookies approaches. However, when the box is loaded, the combination of syncache and syncookies outperforms a pure syncache configuration. Again, as there are no TCP retransmits occurring, the performance difference is not due to entries getting dropped from the syncache hash buckets. This also indicates that the bucket depth of 30 entries that is used in these tests is sufficient to handle the RTT across the local LAN; connections are getting established before they are dropped.

The difference between the two algorithms could be explained by the difference in ISS generation, or by the fact that the standalone syncache needs to perform FIFO drop for a bucket, which is bypassed when syncookies are in use. However, it is not expected that the list management requirements, which consist of few TAILQ_* calls, would be significant. The investigation into the performance difference is still ongoing.

In comparison to the unmodified system presented in Figure 2, there is a dramatic improvement. In this experiment, clients were able to connect to the server and perform useful work (reading one byte), with all attempts completing within 1 second. In the unmodified system, 90% of the connections still had not completed the TCP handshake after 1 second. Even with reduced queue depths, the performance of the unmodified system does not match the new code.

7 Previous Work

David Borman wrote a patch for BSDi which implemented a SYN cache in October 1996, which was released as an official BSDi patch [2]. This implementation used the cache only as a fallback mechanism in case the listen queue overflowed, and did not retransmit the SYN,ACK to the peer. The justification given was that since the host was under attack, performing retransmits would be a waste of CPU time [3].

This code was incorporated into NetBSD[5] in May

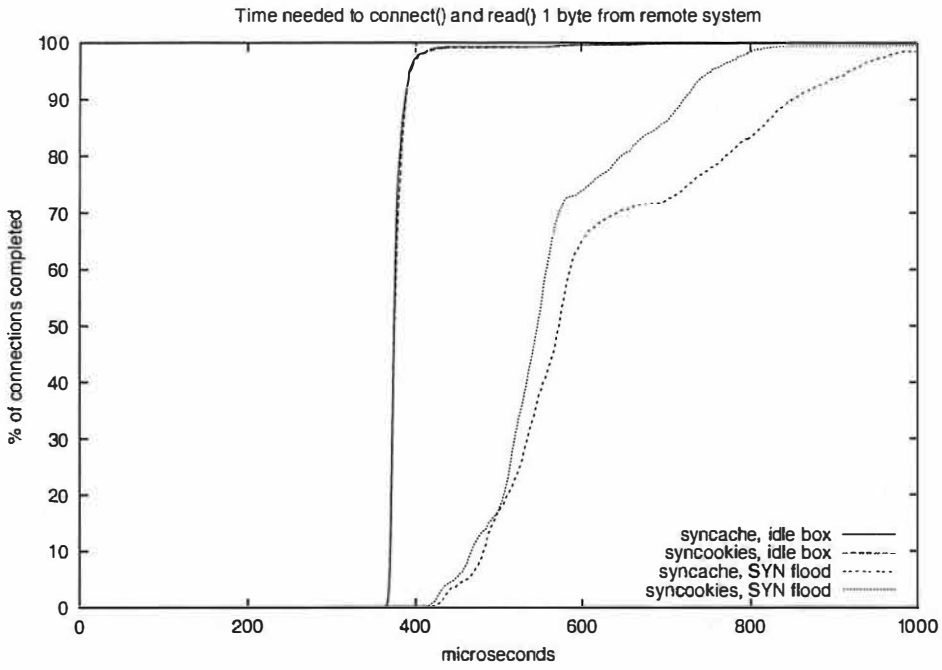

Figure 6: Time required to connect() to remote system and read() one byte in response. No errors at the system call level were observed during the test.

1997 and subsequently enhanced to perform retransmits, as well as having the cache handle all incoming connections, instead of only those which overflow the listen queue. The implementation described in this paper bears a strong resemblance to their existing code.

An alternate approach was taken by Linux, which chose to incorporate syncookies[1] as their defense against this style of attack. On these systems, the syncookie defense mechanism engages only when the normal listen queue overflows.

8 Further Work and Conclusion

When syncookies are enabled, the existing code does not drop any entries from the syncache, choosing to send a syncookie response instead. However, in practice this leads to the syncache being full of bogus entries from a SYN flood, and forces all legitimate connections to be handled by syncookies. Essentially, the system ends up behaving as if there is no syncache, which is not an ideal situation.

An alternate approach that may prove feasible is to use a syncookie as the ISN for all connections, instead using arc4random() in the syncache case. This would permit the replacement mechanism of entries within the syncache to operate as normal, as the returning ACK could be accepted by either by virtue of passing the syncookie check, or by matching an existing syncache entry. This approach is currently under investigation; one issue that needs to be addressed is whether the reduced entropy of a syncookie ISN provides adequate protection from remote attackers as compared to one from arc4random().

In this paper, we have seen that an unmodified machine provides unacceptable response times under a simple I0Mb/s SYN flood attack. Two approaches to handing this load are presented and evaluated, and we show that both are able to extensively migitate the effects of a SYN flood and allow the system to continue operating. This goal is reached by the dual approach of reducing memory consumption and state on the server side, and the use of better algorithms to handle a large number of incompleted connections. With the new code, the same hardware is now able to withstand a SYN flood attack while maintaining an acceptable level of service to legitimate clients.

References

[1] BERNSTEIN, D. J. Syn cookies. `http://cr.yp.to/syncookies.html`.

[2] BORMAN, D. Bsdi implementation of syn cache. `ftp://ftp.bsdi.com/private/44-syn-diffs.gz`.

[3] BORMAN, D. tcpip-impl posting. http://www.kohala.com/start/borman.97jun06.txt.

[4] Freebsd, tuning(7) man page. http://www.FreeBSD.org/cgi/man.cgi?query=tuning&apropos=0&sektion=7&manpath=FreeBSD+4.4-RELEASE&format=html.

[5] Netbsd. http://www.netbsd.org/.

[6] Rfc 1644. Transactional TCP.

[7] STEVENS, W. R. Tcp/ip illustrated.

Flexible Packet Filtering: Providing a Rich Toolbox

Kurt J. Lidl
Zero Millimeter LLC
Potomac, MD
kurt.lidl@zeromm.com

Deborah G. Lidl
Wind River Systems
Potomac, MD
deborah.lidl@windriver.com

Paul R. Borman
Wind River Systems
Mendota Heights, MN
paul.borman@windriver.com

Abstract

The BSD/OS IPFW packet filtering system is a well engineered, flexible kernel framework for filtering (accepting, rejecting, logging, or modifying) IP packets. IPFW uses the well understood, widely available Berkeley Packet Filter (BPF) system as the basis of its packet matching abilities, and extends BPF in several straightforward areas. Since the first implementation of IPFW, the system has been enhanced several times to support additional functions, such as rate filtering, network address translation (NAT), and traffic flow monitoring. This paper examines the motivation behind IPFW and the design of the system. Comparisons with some contemporary packet filtering systems are provided. Potential future enhancements for the IPFW system are discussed.

1 Packet Filtering: An Overview

Packet filtering and packet capture have a long history on computers running UNIX and UNIX-like operating systems. Some of the earliest work on packet capture on UNIX was the CMU/Stanford Packet Filter [CSPF]. Other early work in this area is the Sun NIT [NIT] device interface. A more modern, completely programmable interface for packet capture, the Berkeley Packet Filter (BPF), was described by Steve McCanne and Van Jacobson [BPF]. BPF allows network traffic to be captured at a network interface, and the packets classified and matched via a machine independent assembly program that is interpreted inside the kernel.

1.1 BPF: An Overview

BPF is extremely flexible, machine independent, reasonably high speed, well understood, and widely available on UNIX operating systems. BPF is an interpreted, portable machine language designed around a RISC-like LOAD/STORE instruction set architecture that can be efficiently implemented on modern computers.

BPF only taps network traffic in the network interface driver. One important feature of BPF is that only packets that are matched by the BPF program are copied into a new buffer for copying into user space. No copy of the packet data needs to be made just to run the BPF program. BPF also allows the program to only copy enough of a packet to satisfy its needs without wasting time copying unneeded data. For example, 134 bytes is sufficient to capture the complete Ethernet, IP, and TCP headers, so a program interested only in TCP statistics might choose to copy only this data.

A packet must be parsed to determine if it matches a given set of criteria. There are multiple ways of doing this parsing, but a great deal of it amounts to looking at a combination of bits at each network layer, before the examination of the next layer of the packet. There are multiple data structures designed for efficient representation of the parsing rules needed to classify packets. BPF uses a control flow graph (CFG) to represent the criteria used to parse a packet. The CFG is translated into a BPF machine language program that efficiently prunes paths of the CFG that do not need to be examined during the parsing of a packet.

Ultimately, a standard BPF program decides whether a packet is matched by the program. If a packet is matched by the program, the program copies the specified amount of data into a buffer, for return to the user program. Whether or not the packet was matched, the packet continues on its normal path once the BPF program finishes parsing the packet.

BPF also has a limited facility for sending packets out network interfaces. BPF programs using this facility must bind directly to a particular network interface, which requires that the program know what interfaces exist on the computer. This allows for sending any type of network packets directly out an interface, without regard to the kernel's routing table. This is how the `rarpd` and `dhcpd` daemons work on many types of UNIX computers.

BPF, as originally described, does not have a facil-

ity for rejecting packets that have been received. BPF, although described as a *filter*, can match packets, copy them into other memory, and send packets, but it cannot drop or reject them.

2 Motivation

The need for a powerful and flexible packet matching and filtering language had been evident for a long time. The basic ideas for the BSD/OS IPFW system were the result of several years of thought about what features and functions a packet filtering system must provide. Having highly flexible packet filtering for an end system would be mandatory, and that same filtering system should be applicable for filtering traffic that was being forwarded through a computer.

The immediate need for a flexible packet filtering framework came from a desire to run an IRC client in a rigidly controlled environment. This environment consisted of a daemon that could be run in a chroot'd directory structure, as well as a highly restrictive set of packet filters. These filters could not just prevent unwanted inbound packets, but perhaps more importantly, could also discard unwanted outbound packets. The BSD/OS IPFW system was thus originally intended to filter both the inbound and outbound traffic for a particular host.

Many of the most popular contemporary packet filtering systems of the initial design era (circa 1995) were incapable of filtering packets destined for the local computer or originating from the local computer. The available filtering systems concentrated on filtering traffic that was being forwarded through the computer. Other major problems with the existing packet filtering systems were the inability to do significant stateful packet forwarding and unacceptably low performance [screend]. `screend` does keep track of IP fragments, which is a limited form of stateful packet filtering.

Further motivation for a flexible packet filtering system was the lack of any other standard packet filtering in the stock BSD/OS system of that era. There was customer demand for a bundled packet filtering system, which was not fully met by the other widely available packet filtering systems [ipfilter]. `screend`, the most widely available contemporary packet filtering system, provided many good lessons in packet filtering technology. The BSD/OS IPFW system was designed with the lessons learned from `screend` in mind.

A consideration for the implementation of a new, flexible packet filtering framework was the realization that as the Internet grew, the number of attacks from other locations on the Internet would also grow. Having a powerful matching language tied to the filtering capabilities would allow for a single BSD/OS computer acting as a router to protect any other computers behind the filter.

3 Need for Flexibility

An early design decision for IPFW was that the system should present as flexible a matching and filtering framework as possible. As few filtering rules as possible should be directly embedded in the kernel. As much as possible, filtering configuration and policy should be installed into the kernel at runtime, rather than compiled into the system. This decision has reaped many benefits during the lifetime of this system. Because IPFW is extremely flexible, it has been applied to many problems that were not in mind at the time it was designed. To borrow Robert Scheifler's quote about the X Window System Protocol, IPFW is "intended to provide mechanism, not policy." [RFC1013]

4 Other Packet Filters

As was noted earlier, there were several other packet filtering technologies when IPFW was first envisioned. In the years since, other filtering technologies have been developed, some specific to a particular operating system and others available on a variety of platforms. A comparative analysis with these other packet filters allows one to more fully appreciate the flexibility of IPFW.

One of the most important differences between IPFW and these other filtering systems is that IPFW actually downloads complete programs to be evaluated against the packets. The other filtering systems are all rules-based. By evaluating an arbitrary program, an entirely new methodology of packet filtering can be installed without rebooting the system. In a rules based system, any new type of rules requires code changes to the filtering system, as well as a reboot to make it active. Dynamic loadable kernel modules can approximate the program download facility as modules could be replaced with new filtering rule capabilities without requiring a system reboot.

4.1 Darren Reed's ipfilter

The ipfilter package is available on many versions of many UNIX-like operating systems, from BSD/OS to older systems such as IRIX to small-footprint systems like QNX to frequently updated systems like FreeBSD. It supports packet filtering, provides a Network Address Translation (NAT) implementation, and can perform stateful packet filtering via an internal state table. Like the other examined packet filters, ipfilter is a rules based system. It can log packet contents to the pseudo-device "ipl." [ipfilter] [ipfilterhowto]

4.2 FreeBSD's ipfirewall System

FreeBSD provides a packet filtering interface, known as ipfirewall. This system is often referred to as `ipfw`, which is the name of the management command.[1] This is a rules based packet filtering mechanism, which is manipulated internally by socket options. There is an additional kernel option (`IPDIVERT`) to add kernel divert sockets, which can intercept all traffic destined for a particular port, regardless of the destination IP address inside the packet. The divert socket can intercept either incoming or outgoing packets. Incoming packets can be diverted after reception on an interface or before next-hop routing. [FreeBSD]

4.3 Linux 2.2: ipchains

The Linux ipchains implementation provides three different services: packet filtering, NAT (called masquerading), and transparent proxying. The packet filtering capabilities are based on having "chains" of rules, which are loaded at three different filtering locations: input, forward and output. Each "chain" location can have multiple rules appended, inserted or deleted from that location. The rules are relatively simple and allow for chaining to another named rule if a particular criteria is matched. Arbitrary data inspection of packets is not permitted. [ipchains]

4.4 Linux 2.4: iptables

The Linux iptables implementation (sometimes referred to as "netfilter") is a complete rewrite and extension of the ipchains filtering system. Substantial cleanup and fixing of multiple idiosyncrasies in handling how packets destined for the local computer are processed have been made. Support for stateful packet filtering has also been added to the system. The command line syntax for specifying packet headers for each rule has been changed since the ipchains release. A `QUEUE` disposition for a packet has been added, which specifies that the packet will be transferred to a user process for additional processing, using an experimental kernel module, `ip_queue`. [iptables]

4.5 OpenBSD's "pf" System

OpenBSD 3.0 includes pf, a packet filter pseudo-device. As a rules-based filter, users are restricted to the available set of rules included with pf. Manipulation of the pf pseudo-device is managed through the `pfctl` command. Internally, the system is controlled by `ioctl` calls to the pf device. Rules can be applied on an *in* or *out* basis, and can be tied to a specific interface as well. As a very new packet filtering mechanism (it was written from scratch, starting in June 2001) it does not have an established track record, and is still undergoing change. [OpenBSD]

4.6 TIS Firewall Toolkit

The TIS Firewall Toolkit (fwtk) and other proxy firewalls not only examine the source and destination of packets, but also the protocol being sent. New application proxies that understand the protocol must be written for each new type of service. While this approach does allow for additional levels of security as the proxy can watch for attack methods that exploit a particular protocol, it requires a much deeper understanding of each new protocol before filtering that type of traffic. [fwtk]

5 Design Elements

Several elements of the overall design and implementation of the BSD/OS IPFW system are worth a detailed examination. Some of the more interesting design choices are discussed below.

5.1 BPF Packet Matching Technology

Because of the many fine matching properties of BPF system, as noted in Section 1, it was selected as the core technology for packet matching and classification in the BSD/OS IPFW system.

5.2 Download Filter Programs into Kernel

The concept of downloading filters into the kernel was not a novel idea. The IPFW author was familiar with a few obscure packet filter technologies that had the filter coded directly into the network stack.[2] While highly inflexible in operation, this type of filter system did make an attacker work harder when attempting to subvert or weaken an installed filter. The marginal security benefit of a filter compiled into the kernel was dwarfed by the numerous advantages of a downloadable packet filter. Early versions of IPFW had the ability to both password protect filters as well as make downloaded filters immutable. Both of these features were eventually dropped as the additional security provided only came into effect once the computer running the filter was compromised. Once the computer has been compromised to that extent, the added security was not considered to be valuable enough to warrant the costs of maintaining the implementation.

5.3 IPFW Kernel Socket

Prudent reuse of kernel facilities is always a goal when designing a new subsystem for the UNIX kernel. The BSD/OS IPFW system needed a method for transmitting data about packets and filter programs from the

kernel to programs running in userspace. In some other historic packet filters, this would have been done via the `ioctl` system call, which requires some artificial file to open. Adding a new system call for this purpose might be justified, but every new system call is generally viewed with suspicion.

Instead of adding a new system call, a new instantiation of a kernel socket was made. A new pseudo-IP protocol was defined, which is accessed via a raw internet domain socket. Because sockets were defined to provide an efficient mechanism of moving streams or packets of data to and from the kernel, they are appropriate for the task of moving data about packet filtering to a user application. In the case of the IPFW socket, the data is always generated by the kernel.

The raw IPFW socket provides important functionality in a standard interface with which programmers are familiar. The socket interface also provides for zero (or many) readers of the data. An IPFW filter can send packets or data about packets it has matched back to a userspace program, regardless of the final disposition of the packet. The userspace program may then log the packet, or it might further process a rejected packet, including re-insertion of a possibly modified packet back into the network via a raw IP socket.

High precision timestamps, in the form of a `timespec` structure, are available on packets read from the kernel socket, if the user has requested them. This timestamp is added during the logging operation, so the user application does not have to worry about getting an accurate timestamp when it reads the packets from the socket.

IPFW uses the `sysctl` system call to pass information about filter programs back and forth between the kernel and userspace programs. `sysctl` is used to copy the filter programs into the kernel as they are installed. It is also used for gathering statistics about the IPFW filters installed on the computer. The `sysctl` interface is another example of a flexible programming paradigm. It provided a natural expression of hierarchy that was easily expandable, did not require artificial files to open and reused an existing kernel interface.

5.4 Multiple Filtering Points

One of the unique features of the BSD/OS IPFW system at the time it was designed was the inclusion of multiple filtering points in the kernel. The original BPF system only allowed for tapping of packet traffic at each physical interface. The BSD/OS IPFW system provides five logical points where filters may be installed in the kernel.

Table 1 lists each filtering location in the kernel. Each filtering location has an associated default action. When a filtering location has at least one filter installed, if no explicit disposition for a packet is provided by the filter, the default action will be applied to the packet. The passage of packets through the various filtering locations is described in detail in Section 6 of this paper.

Location	*Modify?*	*Default Action*
pre-input	yes	accept
input	no	reject
forward	yes	reject
pre-output	yes	accept
output	no	reject

Table 1: IPFW Filtering Locations

5.5 Stackable Filters

Each filtering point in the kernel is actually the attachment point for a stack of filter programs. Filter programs can easily be pushed onto the stack, popped off the stack, or inserted into the middle of the stack for each filtering point. Individual filters each have a priority (a signed 32 bit number) that determines where the in the stack the filter is actually placed. Multiple filters installed at the same priority, at the same filtering location, operate as a traditional stack.

Filters may also have a symbolic tag to aid in their identification, replacement, or deletion.

5.6 Flexibility of Actions

After classifying a packet according to whatever rules are in place, a packet filtering system has to perform an operation on the packet. A simple packet filtering system has just two operations, "accept" and "reject." The BSD/OS IPFW system has three additional operations. The `log` action takes a specified amount of the packet and copies it to the IPFW kernel socket. The `call` action allows the current packet to be passed to a different named filter for further processing. The `next` action calls the next filter in the stack of filters installed at the current filter location. In addition, the BSD/OS IPFW system allows packets to be modified explicitly by the filter program, or as the consequence of calling another filter program. The classic "accept" and "reject" actions have been extended so they can also optionally log the packet to the kernel socket.

5.7 Filter Pool

In addition to the explicit filtering points in the kernel a pool of filter programs can be installed into the ker-

nel, not associated with a particular filtering point. This allows common filter programs to be installed into the filter pool and then be referenced from any of the other filters installed in the running system. Currently only BPF based filters have the ability to call a filter from the pool. The filter called may delete the packet or return a value associated with the packet. Typically this value is boolean. The called filter might also be used to record some state that can later be accessed.

Unlike BPF programs, it is possible to create an infinite loop of called filters. There is no loop detection in the filter software, which could be considered a flaw. Users of the IPFW system are obligated to understand the interactions between all their filter programs.

5.8 Circuit Cache

Although BPF filters themselves are stateless, by using custom coded filters, such as the circuit cache, the filters can access saved state about a connection. The circuit cache provides the system with two features. The first is the ability of a BPF program to request the circuit described by a packet be added to the cache. A circuit is defined as the combination of the source and destination addresses, along with the source and destination ports for the upper level protocol, if relevant. The second is the ability to pass a packet to the cache for it to determine if that session has been seen before. For example, TCP packets can be divided into "Initial `SYN`" packets and "Established" packets. Initial `SYN` packets are subject to potentially complicated rules to determine if the session should be allowed. If the packet is to be accepted, it is passed to the circuit cache asking for an entry to be added for its circuit. Any Established packet is simply passed to the circuit cache for query. If the packet does not match an existing session, it is rejected. The circuit cache understands the TCP protocol and when caching TCP circuits it can optionally monitor `FIN` and `RST` packets and automatically terminate a circuit when the TCP session is shut down. Circuits may also automatically be timed out to reclaim kernel resources after a configuration period of inactivity.

5.9 Custom Coded Filters

While BPF-based filters are the most flexible and commonly used filters within the BSD/OS IPFW system, they are not the only method of defining a filter. There are a variety of custom coded filters available. Custom coded filters are C modules that are compiled into the kernel. These typically provide a very rigid set of filtering capabilities. Some non-BPF filters included with IPFW can be used to write traditional, rules based filters. These non-BPF filters may not be able to make use of the advanced features of IPFW due to limitations in their design. Several examples of custom coded filters are described in Section 10 of this paper.

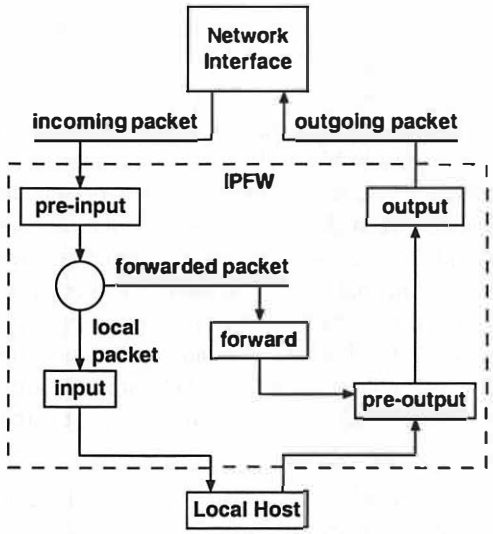

Figure 1: IPFW Filtering Locations

5.10 Transparent Proxying

IPFW's ability to force any packet to be delivered to the local computer allows for the creation of transparent proxies for multiple services. An additional small change to the TCP stack in BSD/OS complements this ability. The `SO_BINDANY` socket option allows a program to listen on a particular port, and bind to whatever IP address for which the connection request was originally intended. This happens regardless of whether the IP address is bound to one of the computer's interfaces. This support makes writing transparent proxies straightforward.

6 How it Works

IPFW operates on Internet Protocol (IP) packets that are received or sent by the computer running IPFW. In general there are three types of packets: packets that were sent to the computer, packets that were generated by the computer, and packets for which the computer is acting as a forwarder.

When a packet arrives on the computer (the packet is either sent to this computer or this computer is forwarding the packet), the network driver copies that packet into an mbuf. If the packet is an IP packet, it is placed on a queue of IP packets to be processed by the kernel. The interface on which the packet arrived is also recorded in the mbuf and can be retrieved by any called IPFW filter.

The `ip_input()` routine in the kernel then dequeues the packet, performs sanity checks on the packet and determines the destination for the packet. If the destination is the local computer, the kernel will perform packet reassembly. IP packets may be broken into smaller packets (fragmented) if a network element in the path between the source and destination is not able to handle the entire packet as a single datagram. Finally, once the packet is complete, the kernel will queue the packet on the correct IP protocol queue (such as TCP or UDP). Packets that are to be forwarded are not re-assembled. These packets are sent on to `ip_forward()` and eventually on to `ip_output()` for transmission to the destination.

When IPFW is used, `ip_input()` will call the pre-input filter chain, if present, just after performing basic sanity checks. This filtering is performed prior to determining the destination of the packet. Because very little examination of the packet has been performed, and no extra state about the packet is stored in the kernel, it is safe for the IPFW filter to modify the packet contents. It may even force a packet to be delivered to the local computer, even if the destination address does not match the address of any of the interfaces on the computer. The most basic modification is to delete the packet, which causes `ip_input()` to stop processing the packet. Allowing modification of any type at this point allows for various specialty filters such as NAT and packet reassembly. Packet reassembly can be performed explicitly by calling the "rewrite" named filter. Packet reassembly is useful so that following filters will always see complete IP packets and not IP fragments. The ability to modify the packet is the reason that the pre-input filter point was added to IPFW.

Once any filters on the pre-input filter point have been executed, `ip_input()` continues with normal processing, which is to determine the destination of the packet. If the packet is to be delivered locally, then processing continues normally up until the point where the packet would be queued for an upper level protocol. The fully formed packet is passed to the input filter chain. No modification of the packet contents is allowed at this point as significant sanity checks have been performed on the packet. The packet may still be dropped, logged, or both dropped and logged. Once the packet completes the input filter processing, it is either discarded (rejected) or queued for an upper level protocol as normal.

Received packets that are not to be delivered locally are to be forwarded and are passed to `ip_forward()`. The `ip_forward()` routine determines if the packet can be forwarded by the computer. This decision is made by ensuring a route exists for the destination address and the packet's time to live has not expired. The packet is then passed to the forward filter chain. The forward filters have access to the interface indexes for both the input and probable output interfaces. It is possible for the output interface to change between `ip_forward()` and `ip_output()`, though typically this is not the case. Knowledge of the input and output interfaces provides assistance in filtering packets with spoofed addresses. The forward filter, like the pre-input filter, is allowed to modify the packet. The main restriction on modifications is that a forwarded packet should not be modified into a local packet. The packet should either still be destined for an external computer after modification or it should be deleted. Once the forward filter chain has been called the rest of `ip_forward()` is executed and eventually the packet is passed on to `ip_output()`.

Packets passed to `ip_output()` are either locally generated or being forwarded through this computer. In both cases, `ip_output()` verifies that a route exists for the destination address and (re)determines the destination interface for the packet. The pre-output filter chain is then called. This filter, much like the pre-input filter, may modify the packet. In addition, it may specify a different IP address to be used for the next-hop routing of this packet. This override of the next-hop routing destination is done through an out-of-band mechanism. This capability allows the pre-output filter to actually determine which interface the packet should be sent out when there are multiple possible output interfaces. If an IP address is provided via the out-of-band method, or the destination IP address inside the packet is changed, the routing lookup is repeated. The pre-output filter is not called a second time.

For forwarded packets, all filtering is now complete. For packets that were locally generated the output filter chain is called immediately after the pre-output filter. Like the input filter chain, the packet may not be modified by the output filter chain. The `ip_output()` routine will eventually call the network interface's output routine. If IPFW rate filtering (as discussed in Section 10) is being used, the `ip.rateoutput()` routine is actually called instead of the interface's output routine. The `ip_rateoutput()` routine is responsible for eventual delivery of the packet to the network interface or dropping of the packet.

7 BPF Language Overview

The most used and most flexible filter type in IPFW is the BPF filter. As mentioned earlier, this type of filter uses the BPF pseudo-machine. The BPF pseudo-machine has been enhanced for use with IPFW. Only one

totally new BPF instruction was added for IPv4 packet processing. A new memory type was added, as well as the ability to modify the packet being processed. IPv6 enhancements have been added and are discussed at the end of this section.

The new BPF instruction, CCC, enables the calling of a filter on the "call filter chain." While it might seem that the acronym stands for "Call Call Chain," it was actually derived from "Call Circuit Cache." The circuit cache was the reason for the creation of the call chain. The CCC instruction returns the result of the call in the A register.

The new memory type is called ROM and is an additional memory area to the original BPF memory spaces. The original memory spaces included the packet contents as well as the scratch memory arena. While the first implementation did in fact store read only information, the term ROM is now a misnomer as the ROM locations can be modified by the filter. This space, called "prom" in the source code, is used to pass ancillary information in and out of the BPF filter.

While the `bpf_filter()` function does not have any innate knowledge of the meaning of these memory locations, IPFW assigns meanings to several locations:

0 IPFWM_AUX	An auxiliary return value (for errors)
1 IPFWM_SRCIF	The index of the source interface (if known)
2 IPFWM_DSTIF	The index of the destination interface (if known)
3 IPFWM_SRCRT	The index of the interface for return packets
4 IPFWM_MFLAGS	The mbuf flags
5 IPFWM_EXTRA	Bytes of wrapper that preceeded this packet
6 IPFWM_POINT	What filter point was used
7 IPFWM_DSTADDR	New address to use for routing to destination

The BPF filter is intelligent about setting these values. As some of these values, such as IPFWM_SRCRT, can be expensive to calculate, the filter is examined when passed into the kernel. A bitmap is built of all ROM locations referenced by the program and only those locations are initialized.

In order to support the ROM memory space, the calling convention of the `bpf_filter()` function was changed to pass three additional parameters:

```
int32_t *prom;  /* ptr to ROM memory */
int promlen;    /* count of valid bytes */
                /* in the memory space */
int modify;     /* boolean to indicate */
                /* whether packet */
                /* can be modified */
                /* by bpf_filter() */
```

All existing calls to `bpf_filter()` were modified to pass NULL, 0, 0 for these three values.

IPFW has been adapted for use with IPv6. This work was implemented with the NRL version of IPv6. More recent releases of BSD/OS use the KAME IPv6 implementation. The changes to support IPFW in the KAME IPv6 stack have not yet been written.

In order to support IPv6, several other new enhancements were made to the BPF pseudo-machine. Triple length instructions were added. A "classic" BPF instruction is normally 64 bits in size: 16 bits of opcode, two 8 bit jump fields, and a 32 bit immediate field. A triple length instruction has 128 bits of additional immediate data (the length of an IPv6 address). A new register, A128, was also added. The load, store, and jump instructions now have 128 bit versions. The scratch memory locations have been expanded to 128 bits, though traditional programs only use the lower 32 bits of each location. An instruction to zero out a scratch memory location (ZMEM) was added. Because BPF was not extended to handle 128 bit arithmetic, a new jump instruction was created that allowed for the comparison of the A register to a network address, subject to a netmask. The netmask must be specified as a CIDR style netmask, specifically a count of the number of significant bits in the netmask.

ROM locations only have 32 bit values and it is in the ROM that a new destination routing address is passed. Currently it is not possible to use the next-hop routing capability with IPv6.

8 IPFW Filtering Language

Initially BPF filters were written in BPF assembly[3] with the aid of the C pre-processor (cpp). It was thought that many assembly fragments would be written for various needs and that the final filter would include these fragments. It was quickly determined this was not a very user friendly way of programming filters. It yielded opaque filters such as:

```
// IP header length into X
ldx  4 * ([0] & 0xf)
// Protocol of packet
```

```
    ld   [9 : 1]
    // Is it UDP?  Jump to L1 if not
    jeq  #17 - L1
    // Move ip length into A
    txa
    // Add 8 bytes to skip UDP header
    add  #8
    jmp  L3
L1:
    // Is it TCP?  Jump to L2 if not
    jeq  #6 - L2
    // Load TCP flags into A
    ld   [x + 13 : 1]
    // Jump to L11 if SYN bit is set
    jset #2 L11 -
    // If SYN is not set, just accept it
    ret  #IPBPF_ACCEPT
L11:
    // Move ip length into A
    txa
    // Add 20 bytes to skip TCP header
    add  #20
    jmp  L3
L2:
    // Just move ip header length into A
    txa
L3:
    or   #(IPBPF_ACCEPT | IPBPF_REPORT)
    // Accept the packet and report it
    ret  A
```

A new language was clearly needed.[4] Existing filtering languages were of little help as they were rules based and not programmatic. The ability to use the programmable features of BPF was a key design goal of IPFW. Since BPF does not allow reverse jumps, there is no facility for loop constructs. This results in two possible constructs: a sequence of instructions, and if/then/else clauses. The IPFW filtering language was designed with this in mind. The general form of the language is:

```
condition {
    true action
} else {
    false action
}
```

The false action is optional and typically omitted in normal filter programs. Note that "if" is implied. Initially "else" was also implied, however, this reduced readability so it was added back into the language.

In addition to this generic construct, there is a block statement, which is essentially a series of "if" and "else if" statements. There is also a "case" statement which is similar, but not identical, to a C "switch" statement.

Most actions are either another construct or a terminating condition, such as "accept" or "reject."

9 End-User's Perspective

From the end-user's perspective, creating a packet filter involves writing a text file that contains the filter, compiling the filter from the command line, and loading the compiled filter into the kernel from the command line. A user could create the following sample filter in a file called `forward`:

```
#define SERVER   192.168.1.10
#define MAILHOST 192.168.1.15

switch ipprotocol {
case tcp:
    // TCP packets should never come in
    // as fragments
    ipfrag {
        reject;
    }
    // TCP packets need at least 20 bytes
    iplen (<20) {
        reject;
    }
    // We just accept established
    // connections
    established {
        accept;
    }
    // Allow incoming services to
    // some computers
    switch dstaddr {
        case SERVER:
            dstport(ssh/tcp, telnet/tcp,
                    ftp/tcp, http/tcp) {
                accept;
            }
            break;
        case MAILHOST:
            dstport(smtp/tcp) {
                accept;
            }
            break;
    }
    // All other requests are rejected
    // and logged to the kernel socket
    reject[120];
    break;

case udp:
    // Accept non-first fragments
    ipfrag && !ipfirstfrag {
        // But don't allow fragmented
        // UDP headers
        ipoffset(<8) {
            reject;
```

```
        }
        accept;
    }
    // UDP packets need at least 8 bytes
    iplen(<8) {
        reject;
    }
    // We just accept all UDP packets
    accept;
    break;

case icmp:
    // We just accept all ICMP packets
    accept;
    break;

default:
    // We reject any other protocols
    // and log them to the socket
    reject[120];
}
```

The user could then compile and load the filter on the forward location in the kernel:

```
# ipfwcmp -o /tmp/ipfw.forward forward
# ipfw forward -tag fwd-filt \\
       -push /tmp/ipfw.forward
```

If the user wanted to examine the effectiveness of their filter program, they could:

```
# ipfw forward -stats
forward filter statistics:
   3068169 packets rejected
           3033113 reported
  19496389 packets accepted
                 0 reported
        14 errors while reporting
         0 unknown disposition
```

10 IPFW Specialty Filters

The fact that IPFW is a general filtering framework allows very specialized filters, written in C, to be linked into the kernel. Some examples of this are NAT, IP flow monitoring, and rate filtering. Since IPFW filters have the ability to call other filters, it is possible to use an IPFW filter to do the bulk of the work, but still use a fast C-based hashed lookup scheme on a large pool of addresses.

Rate filtering provides a mechanism that controls how quickly packets are allowed to leave a computer. Different classes of packets can be assigned different rates. Each class of packets is determined by an IPFW classification filter. For example, it is easy to create a class for all outbound http traffic and assign a particular bandwidth limit to it. Additional rate classes could be defined for other protocols and different bandwidth limits applied to each class.

Protocol rate filtering is used in conjunction with a modified circuit-cache to impose a rate limit on individual remote hosts, rather than on a class of packets leaving a computer. For example, a DNS server may limit the number of requests that a particular client can make in a given time period.

The NAT filter provides IP address and port translation services for TCP and UDP traffic. This transparent filtering provides the usual benefits of network address translation.

Flow monitoring gathers data on TCP and UDP traffic between two computers. A TCP flow is a TCP session, while a UDP flow is a series of UDP packets that share the same source and destination address and port. The flow monitoring facility provides data similar to Cisco's NetFlow implementation. This data is useful for network capacity planning as well as high level network protocol analysis.

11 Real World Examples

IPFW has many potential uses for anyone who has IP connectivity. Given the multitude of potential filtering operations available, it is instructive to see how IPFW is used by three sample installations.

11.1 Home User

The first sample installation uses a small set of the IPFW capabilities. This installation has two network connections, one via a dialup modem using PPP, and a second connection via an IDSL modem. The BSD/OS computer running as a router uses the IPFW capabilities to perform three distinct tasks.

The first is the next-hop routing of outgoing traffic to select between the IDSL and PPP outgoing interfaces, based on the source address of the packet. The second is packet filtering of inbound traffic to the servers located behind the filtering computer. The last is to provide NAT services for other client computers behind the filtering computer.

11.2 Shared Corporate Network

The second sample installation uses a larger set of the IPFW capabilities. This installation has only one upstream connection, but multiple different client networks behind the filtering host. The filtering host is also used

as a filtering gateway between the different client networks.

The filtering host has a forward filter that allows inbound access for a select number of protocols, to a rigidly defined list of servers on the different client networks. This filter is rather complicated, as it must treat each of the client networks attached to the filtering computer with a different set of filtering options, specified for each of the clients.

The filtering host has a pre-input filter that terminates outbound http requests on the filtering computer, so that a transparent http cache can operate for all the different client networks behind this host. There is also an input filter in use to prevent external users from connecting to the http cache. The input location filtering is done to prevent any inbound TCP connections from ever being established directly to the cache daemon.

The filtering host also has a set of rate filters, to limit how much bandwidth certain network protocols (for example, nntp) are allowed to use at any given time. The rate filtering capability has proven very useful for simulating low-bandwidth links for testing during debugging of network tools that must transfer large quantities of streaming information over long-haul networks. Some of the server computers behind the filtering host also use rate filters to limit the amount of outbound http traffic that may be sent. This is used to enforce bandwidth limits to which the clients have agreed.

The filtering host also has a set of filters in place to filter connections from the "nimda" worm first noted in autumn 2001. This worm attacks web servers and attempts 14 different accesses, probing for known security holes. While none of the servers at this location are vulnerable to this worm, the attacks still cause a great number of extraneous log entries to be made on each web server. There are so many log entries from the attack probes that the normal log entries are drowned out in the noise of the attack entries.

Three filters are used to combat this worm. The first is the system provided, custom filter called "rewrite," which allows sending RST packets to one end of a TCP connection. The second filter is a custom, hand-coded BPF assembler filter that is installed into the filter pool with the name "src_dst_swap." This filter swaps the source and destination IP addresses in a packet, as well as the source and destination port addresses. The final filter is an IPFW filter that examines the interior of packets destined for the protected clients' web servers. If the filter detects the "nimda" attack signature after the TCP session has gone through the three-way handshake, the filter calls the "src_dst_swap" filter and then calls the "rewrite" filter. The "rewrite" filter sends a TCP RST packet to the internal web server, which will cause it to tear down the just established TCP session. By resetting the TCP connection on only the protected client web server, but not that of the attacking computer, the client web server's kernel resources are immediately freed for reuse. The attacker's kernel resources are intentionally left occupied. By resetting the TCP connection before any data is sent to the web server, no error message is logged by web server. This neatly solves the problem of the extraneous log messages caused by the "nimda" worm. It could be trivially enhanced to deal with other such attacks in the future.

11.3 Corporate Firewall

The last sample installation of the IPFW system is a special purpose corporate firewall. A very successful firewall has been built around the IPFW system to dynamically protect a group of servers from malicious dialup users. It uses the IPFW kernel socket and a simple filter at the forward location to read all TCP SYN packets coming from a list of CIDR blocks that represent the dialup modem pools. A continually updated database of IP addresses assigned to legitimate dialup users is queried to determine if the SYN packet should be allowed. If the packet is permitted, a copy of the packet is sent out a different interface, to the servers to allow establishment of the client's TCP session. Other non-TCP packet filtering is also done on these firewalls using regular packet matching and filtering.

12 Future Enhancements to IPFW

There are always ample possibilities for expanding useful software systems, and IPFW is no exception. Under consideration are structural additions as well as changes to make IPFW more accessible to a larger number of people.

Allow loadable kernel modules to implement "Custom Coded Filters." This would allow for the speed of a native implementation of a complex filter, while preserving the ability to reconfigure the filtering system on the fly.

Implement an in-kernel filter compiler that takes BPF programs as input and generates a native (non-interpreted) version of the downloaded BPF filter for execution.

On-demand logging would make it easier to debug filters. This would also allow the assessment of the cost and benefits of making certain system changes.

IPFW support needs to be added back into the KAME IPv6 protocol stack. The IPv6 support needs to be com-

pleted, so that next-hop routing can be used with IPv6 addresses.

Expansion of the protocol throttling support. This would allow limiting the number of responses for a particular protocol and provide another method for preventing denial of service attacks.

While the programmable nature of IPFW makes it extremely flexible, programming is not a strength of all system administrators. This lack of programming experience means that some filters are inefficient and others do not take full advantage of IPFW features. A graphical user interface could address these issues.

13 Conclusions

The IPFW framework has proven to be extremely flexible. It has been used for purposes never dreamed of in the original design. This is often the hallmark of a good basic design.

IPFW has a complete framework for packet filtering services. It has been extended several times since the original implementation but has never needed to be completely redesigned.

IPFW is very reliable. It has been deployed in applications that have passed terabytes of information between reboots. Machines executing IPFW filters have uptimes in excess of one year, even after multiple filter changes and updates during that time period.

Appropriate documentation is crucial to wide acceptance of a new technology. While the IPFW system has been available for several years and has many powerful and useful features, its acceptance has been slow because of incomplete and opaque documentation.

14 Availability

Additional documentation, as well as some of the filters described in this paper, is available at `http://www.pix.net/software/ipfw/`.

15 Acknowledgments

The contributions of several people are hereby acknowledged in their efforts to make this a better paper. Jack Flory and Mike Karels for numerous conversations about packet filtering prior to the first line of the IPFW code being written. Bill Cheswick for prompting the need for the circuit cache, which led to the ability to chain filters and the ability to call other filters from a filter. Dave MacKenzie, Josh Osborne, and Chris Ross, for reading draft copies and making useful suggestions. Donn Seely, for guiding this paper through the submission and review process. The staff at the USENIX Association, for they make this conference possible.

Notes

[1]Even though FreeBSD's ipfirewall is often referred to as `ipfw` it is unrelated to the BSD/OS IPFW system, except in name.

[2]In order to change the filter, one would have to recompile the kernel and reboot the computer with the new kernel.

[3]When work on coding IPFW started, the author searched for the BPF assembler that was described in the original BPF paper. After learning that no actual assembler was ever written, a standalone BPF assembler was written for IPFW.

[4]A language similar to the IPFW filtering language was independently developed for the Ascend GRF router. While the GRF used BSD/OS as the basis of its embedded operating system, the GRF filtering system was independently developed.

References

[BPF] McCanne, Steve, and Van Jacobson, *"The BSD Packet Filter: A New Architecture for User-level Packet Capture,"* Proceedings of Winter 1993 USENIX Annual Technical Conference. (January, 1993).

[CSPF] Mogul, J.C., and R.F. Rashid, and M.J. Accetta. *"The packet filter: An efficient mechanism for user-level network code."* 11th ACM Symposium on Operating System Principles, November, 1987.

[FreeBSD] The FreeBSD Project. *ipfirewall(4); FreeBSD 4.4-stable Manual Page.* http://www.freebsd.org, June, 1997.

[fwtk] Trusted Information Systems, Inc. *TIS Internet Firewall Toolkit* ftp://ftp.tislabs.com/pub/firewalls/toolkit/, March, 1997.

[ipchains] Russell, Rusty. http://netfilter.samba.org/ipchains/, October, 2000.

[ipfilter] Reed, Darren. http:///www.ipfilter.org/, November, 2001.

[ipfilterhowto] Conoboy, Brendan and Erik Fichtner. *IP Filter Based Firewalls HOWTO*

http://www.obfuscation.org/ipf/ipf-howto.pdf, July, 2001.

[iptables] Russell, Rusty. http://netfilter.samba.org/, August, 2001.

[NIT] Sun Microsystems, Inc. *NIT(4P); SunOS 4.1.1 Reference Manual.* Mountain View, California, October, 1990. Part Number 800-5480-10.

[OpenBSD] The OpenBSD Project. *pf(4); OpenBSD 3.0 Manual Page.* http://www.openbsd.org, July, 2001.

[RFC1013] Scheifler, Robert W. *X Window System Protocol, Version 11.* Cambridge, Massachusetts, June, 1987.

[screend] Mogul, Jeffrey C. *"Using screend to implement IP/TCP security policies."* Technical Note TN-2, Digital Equipment Corporation Network Systems Lab, 1991.

A FreeBSD Based Low Cost Broadband VPN Router for a Telemedicine Application

Gunther Schadow
Regenstrief Institute for Health Care, Indianapolis, IN

Abstract

The author developed a small low cost broadband networking router using FreeBSD to support a telemedicine application. Our router design provides IPsec based virtual private networking (VPN) and quality of service (QoS) arrangements, simultaneously supporting two-way real-time video and audio, camera control, streaming video replay and medical record access over the public Internet using Cable-modem links from physician's homes to the Internet. These routers have been critical for the implementation of the telemedicine project after several other attempts with proprietary "solutions" have failed. FreeBSD's integrated IPsec code, QoS facility, firewall, and its ability to squeeze the entire system down into a file system image of less than 8 MB, had been instrumental for the development of our routers. Above all, however, this project would not have been possible without the availability of the system's source code, the ability to investigate bugs, rapid turn-around in the user-support community, and the ability to add missing features into existing software, which is made possible by the open source software development paradigm.

1 INTRODUCTION

We developed a small low cost broadband networking router using FreeBSD to support a telemedicine application, allowing physicians at their homes to interview patients in a nursing home over videoconference. We are currently conducting a study to test the hypothesis that audiovisual interaction with the patient through videoconferencing may enable the physician to make better decisions and reduce unnecessary referrals to the emergency room. We describe this study elsewhere in greater detail [1].

In this paper we present the design and development of our broadband VPN router that provides IPsec based virtual private networking (VPN) and quality of service (QoS) arrangements, simultaneously supporting two-way real-time video and audio, camera control, streaming video replay and medical record access over the public Internet using Cable-modem links to the Internet. We present this as a case study of a real-world application that requires many still relatively new functions of the Internet suite of protocols and the operating systems implementing it. We include some recommendations that we find would make the pieces fit together better. We also hope that the reader may find useful the description of our technical approach and the references we give.

1.1 Why a Custom Router?

Although funds had been allocated to use T1 connection to the 5 participating physicians' homes, our T1 provider did not install the ordered T1 lines to the physicians' homes for over a year after the contracts had been signed. We therefore had to use the public Internet instead. We found that Cable-Modem Internet service in our area had a sufficient bandwidth (2 Mbit/s downstream) even after the ISP throttled uplink bandwidth from 2 Mbit/s down to 100 kbit/s. We also realize that costs would inhibit scaling up T1 line use beyond the limited scope of this study. Being a very application-oriented project rather than genuinely a networking research and development project, the use of self-designed and open-source operating system driven devices was not initially planned; but soon became as important as to rescue the whole project.

Initial approaches were made with Cisco's PIX firewall in combination with Intel's PRO/100 S network interface cards (NIC) capable to offload encryption from the end-system's CPU. (Offloading was critical because of the high CPU and I/O load from the videoconferencing application.) However it turned out that the software kept crashing and the IPsec modes (tunnel vs. transport) did not allow interoperation with our Cisco PIX firewall. Another attempt with Cisco's broadband access router with IPsec capability disappointed because our configurations and control over the device was taken away by the Cable-Modem head-end as soon as a Cable-Internet link had been established. Other third party solutions did not provide for our specific needs (e.g., QoS.) Thus a self-developed device became reasonable.

1.2 Requirements

Our broadband VPN router addresses the following requirements:

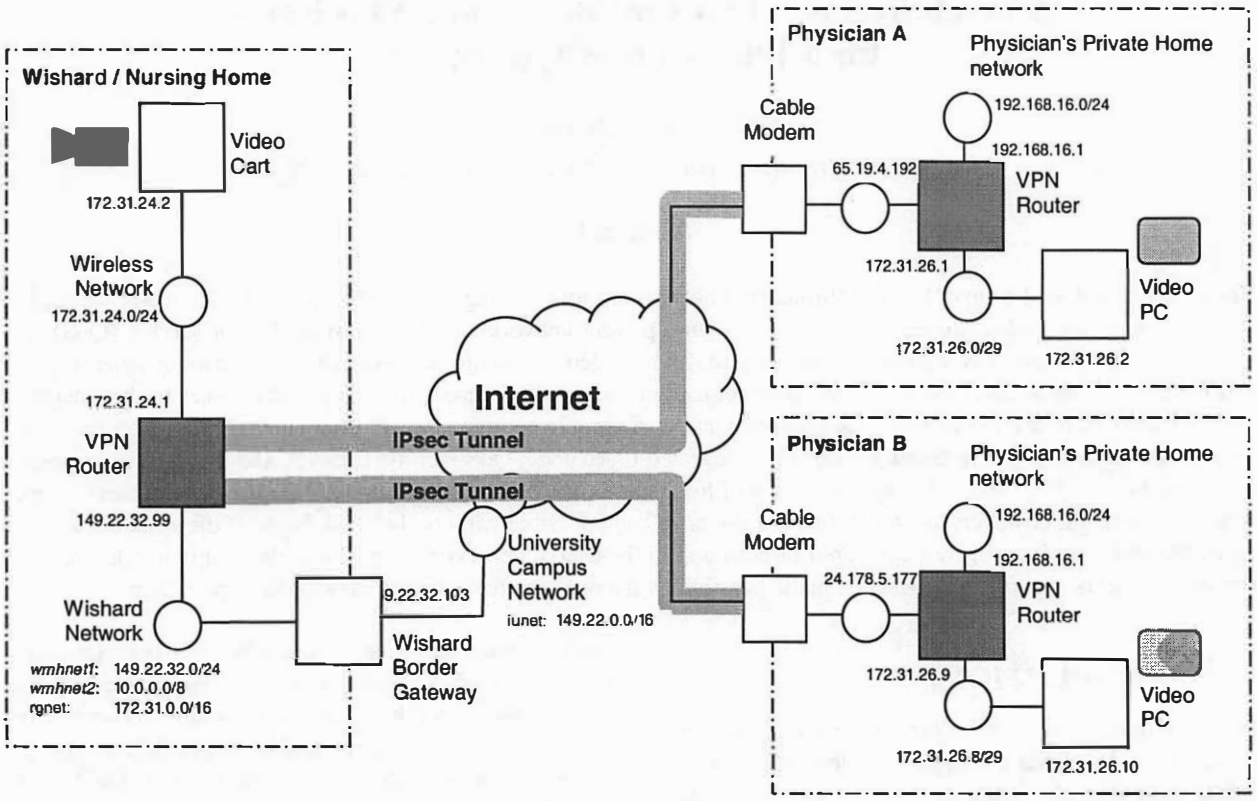

Figure 1: Topology of the Telemedicine Virtual Private Network. IPsec tunnels securely connect remote networks and establish a virtual address structure on top of public IP networks (tthe public IP addresses are disguised.)

a) Route IPv4 packets between a Cable-Modem uplink and a local area network (LAN) in the physician's home.
b) Provide network address translation (NAT) to connect several machines on the LAN to the Internet via the one IP address provided by the Cable-ISP.
c) Secure all communications between the physician's dedicated video conferencing and medical record access computer and the nursing home's LAN using strong encryption at a bandwidth of 2 Mbit/s downstream and 128 kbit/s upstream.
d) Provide a reasonable firewall protection to avoid breaches into the physician's LAN and prevent unauthorized access into the VPN through one of the remote sites.
e) Provide two separate LAN segments (requiring 3 Ethernet interfaces in total) so that the physician's and his family's private use of the Internet is kept separate from office use. In particular, prevent access to the virtual private network from the private segment.
f) Compensate for the asymmetric nature of the Cable-Internet connection with only 128 kbit/s outgoing bandwidth. Coordinate available bandwidth between the essential flows: (1) video (H.261 [2] over UDP, up to 1.5 Mbit/s), (2) audio (UDP, 64 kbit/s, fixed), (3) remote camera control (TCP, low bandwidth but need for low latency and intolerant to packet loss), and (4) medical record access (TCP/HTTP).
g) Be able to route and tunnel Novell NetWare IPX packets used in our (as in many) corporate network.
h) Be small, noise-free, affordable, reliable, and remotely maintainable.

1.3 Overview of the Design

Figure 1 shows the topology of our telemedicine VPN. Physicians are connected in their homes through cable-modems. Each home has one of our VPN routers establishing an IPsec tunnel through the Internet linking to a central VPN router in the nursing home. A portable video cart is connected through a wireless network to this central router.

The VPN is designed as an "overlay network" [3], i.e. using an address space that is distinct from but mapped to the global IP address space through the use of IP aliases to hosts and tunnels between distant physical networks. For example, as Figure 1 shows, our telemedicine network uses a "private" IP address space [4] such as 172.31.0.0/16. Nodes that are only part of that

telemedicine network only have addresses in this network. Nodes that are part of multiple networks have aliases in those networks. Physically distant subnetworks are connected through IPsec tunnels. The IPsec tunnels are established between VPN routers in a star-topology with one central VPN router at the nursing home facility and multiple remote VPN routers, one for each participating physician's home.

A VPN router in the physician's home has three networks attached: (1) the Internet through the cable-modem, (2) the office network that is part of the telemedicine VPN, and (3) the physician's private network that has no route to the VPN.

Packets that travel between the office network and the public Internet are not routed through the tunnel, which would only add load to the tunnel endpoints and increase the physical path lengths.

We developed the VPN routers as a custom device based on a generic UNIX operating system (FreeBSD 4.2-RELEASE) and small generic PC compatible hardware. The assumption was that FreeBSD had all the necessary facilities already integrated and all we needed to do is configure the pieces and compile them into a form that is easy to deploy and maintain. These assumptions have for the most part held true, although we found that making all the pieces play together can be a challenge. We also found bugs and missing features.

2 HARDWARE

Since the hardware market is a fast moving target, much of our writing about our hardware must remain anecdotal, and we make no effort pretending otherwise. The reader may find helpful references to resources. We also believe that most of the issues and tradeoffs we had to face still apply today (approximately 1 year after we made our hardware decisions.)

We required a moderately fast PC compatible with solid state memory of at least 4 MB and dynamic memory of at least 8 MB, 3 network interfaces, small size and simple power requirements at a reasonable price (i.e., given the relative low-end performance of the system, we set an upper limit at $350 per complete system.) Despite these very modest requirements, we could not find a suitable product for over a year.

We monitored the market by tracking the following resources: PC/104 Embedded Solutions [www.pc104-embedded-solns.com] web site and magazine, Linux-devices.com, and the product catalogs of many of the Single-Board-Computer (SBC) vendors such as (in no particular order) Advantech, Tri-M Systems, ICP Electronics, and Diamond-Systems, and others.

Among the trade-offs between features, dimensions, and price the limiting factor was usually the requirement for built-in network devices. Most low-cost/low-end SBCs have no network interface. Many SBCs came with one network interface (usually of the 10 Mbit/s class, 10base-T); however, those would usually also include video and sound interfaces. For us, video and sound interfaces were disadvantaging factors because they would make the product more expensive, larger, consume more power and produce more heat. Furthermore, most "embedded systems" required much additional parts and assembly. Given the difficulty of finding a suitable device with one network interface, finding one with 3 interfaces is nearly impossible in the SBC market. The very few systems we found ranged in the $1000 price class.

An alternative to SBCs, however was a desktop PC-computer of small dimensions with a limited PCI-backplane that allows adding network interface cards. On the FreeBSD "small" mailing list we were hinted to FlyTech's line of small PC products, in which we found the NetPC NC-2 B62 (Intel Celeron, 533 MHz, 64 MB DRAM, 8 MB Disk-on-chip, 1 RealTek 100base-T NIC.) With 2 Intel PRO/100 network interface cards added, and at a total cost of approximately $500, this product was a viable option to implement the project for the scope of the clinical trial. We are still using this FlyTech computer for the central VPN router at the nursing home.

Of particular interest in the low-cost embedded-PC market are so called PC-on-a-chip designs such as the National Semiconductor GEODE, the AMD SC520, and the ZF Micro Devices ZFx86. These chips combine most of the electronics of a common PC motherboard in one chip which dramatically reduces the dimension and power-consumption. They can reduce cost as well, provided that the board design and manufacturing is as cost effective as the mass-production of standard PC motherboards. Having a board custom-designed was not a good option for us, since we only had funds for 30 units to be purchased initially. (The cut-off at which custom design is feasible is at about 100 units.)

For SBC applications, a PC-architecture may not be the smallest or most cost-effective approach. Since in the first months of searching we could not find a reasonable PC hardware we investigated other platforms such as the Intel StrongARM and MIPS. These hardware platforms are supported to a certain extent by both NetBSD and Linux. So, we would have had an avenue of implementing our project.

We finally found a small engineering firm, SOEKRIS Engineering [www.soekris.com] who had designed a device, "net4501," based on the AMD Elan SC520 PC on a chip. This device has 3 100base-T network interfaces on board and a serial interface, no video- and sound. One low-power PCI and one mini-PCI connector

provides the flexibility to accommodate special needs in the field. The SOEKRIS net4501 board comes with a CompactFlash socket for a solid-state memory, which turned out to be cheaper and much easier to work with than the Disk-on-chip devices that most SBC products (including our FlyTech Net-PC) use. The board needs no airflow cooling requiring no moving parts and emitting no noise at all. The hardware is available for approximately $200 including a simple enclosure and a wall-mount AC transformer (enclosure and power supply can be a major cost factor in the deployment of SBCs.)

We had the privilege of testing one of a few samples of the SOEKRIS net4501 board and were convinced that it would be close to the ideal device for our needs. The only problem of this board was that production was delayed due to the economic circumstances and we didn't have the purchasing power to jump-start production. However, a few months later the board did go into production in time for our project to use it.

3 SOFTWARE

We developed the software based on FreeBSD. We choose BSD over Linux because of the KAME IPsec implementation that is becoming a reference implementation. At that time, KAME had just succeeded and partially subsumed other open source IPv6/IPsec projects (WIDE, NRL, INRIA.) Most of these early IPv6/IPsec implementations had BSD as their primary target platform. FreeBSD's built-in support for small systems ("PicoBSD"), its large user base, and focus on stability made it most attractive among the other BSD systems, NetBSD, and OpenBSD.

We begun development on this project based on the 4.0-RELEASE of FreeBSD. We loosely track the FreeBSD RELEASE branch rather than keeping abreast of STABLE or even CURRENT, because having deployed the systems, we can not tolerate very frequent changes as we do not have the resources to test the system with every new build on a weekly or daily basis. However, we did find that the KAME IPsec code available in the RELEASE was not up to the latest critical bug-fixes, hence we usually had to apply a certain KAME "snap-kit" on top of the major FreeBSD release.

We will present the software design along the lines of the major requirements: (1) firewall; (2) network address translation (NAT); (3) IPsec-based VPN; (4) quality of service (QoS) arrangements; and (5) Novell IPX protocol routing and tunneling.

3.1 Firewall

In a VPN, firewalls are essential security elements more so than in a corporate network without VPN. This is because each remote site is mapped into the internal corporate network by means of VPN tunneling but the corporate network administration has less physical control over the remote site. Thus, a remote site can become an open door for intruders into the corporate network if not sufficiently protected against such attacks. Most importantly, the firewall must delete all incoming packets that have a source or destination address into the corporate network.

Several different firewall packages are available for open source UNIX systems. Initially, FreeBSD has a "native" IP-Firewall (IPFW) facility. Even though we found the IPFW design and integration into the system very useful, we did switch to Darren Reed's IP-Filter (IPF) [5] system early on. IPF has the advantage over IPFW of being available for systems other than FreeBSD, NetBSD in particular (as we alluded to above, we planned for migrating to NetBSD had we not found the SOEKRIS hardware in time for the project to continue.) The IPF design and implementation claimed to be more secure (e.g., it allowed updates to the firewall rules without inconsistent temporary states) and better monitored, if only because IPF had a larger user base, (including all BSDs, Linux and even many commercial UNIX systems, such as Solaris, IRIX, and HP-UX.)

To the user, the difference between IPFW and IPF is mostly the different rule syntax. IPFW has a richer set of functionality available for the filtering rules, such as dropping, NATing, forwarding, queuing/delaying. Through the "divert socket" one can tie user-defined special packet handler processes outside the kernel into IPFW (the NAT handler natd(8) being such an extra-kernel process.) Conversely, NAT with IPF are two distinct kernel facilities configured with ipf(8) and ipnat(8) using different configuration files and only a loosely related configuration syntax.

3.2 Network Address Translation (NAT)

We have to perform network address translation (NAT) on both the remote and the central VPN routers. The remote VPN routers perform NAT to allow communication between the office and private networks and the Internet to which the cable-ISP provides only one IP address. The central VPN router also performs NAT for communications of the video cart to the Internet. In addition, the central VPN NATs all communications coming out of the tunnel with destination in a public network (e.g., hospital LAN, university campus network that.)

The same choices as for the firewall exist for NAT: IPFW and IPF. We found that more complex rules could be more easily expressed with IPFW than with IPF. For example, one of our rules is to apply network

address translation (NAT) only to destinations not routed through the VPN tunnel and all destinations if traffic originated on the "private network." With IPFW one can fine-tune every NAT rule exactly for each source and destination and all the other criteria available for filtering, and one can reuse the same NAT resource pool (e.g. a set of mapped port numbers) for multiple rules. Conversely, with IPF's NAT facility, ipnat(8), the criterion language is only a minimal subset of the ipf(8) filter language.

3.3 IPsec

For the beginner, IPsec comes in a confusing number of modes (tunnel/transport, configured/negotiated security associations, pre-shared keys/other authentication) [6] and various alternative ways to configure them with BSD/KAME. In addition, KAME is not the only alternative. Notably, Pierre Beyssac's pipsecd program, implements limited IPsec function using the generic tunnel pseudo-device tun(4). The tun(4) device is a standard part of BSD and Linux and represents a network interface that passes packets to a handler program, which, in the case of pipsecd, can encrypt and encapsulate the packets and pass them on. Finally, not all of KAME is equally well tested and integrated into FreeBSD. All of these factors tend to confuse the issue of getting an IPsec system operational. We will address each of them in this subsection.

ESP, AH, IPCOMP, some or all: The choice of using the Encapsulating Security Payload (ESP) protocol [7] or the Authentication Herader (AH) protocol [8] alone is simple: if encryption is part of the requirements (as it usually is), ESP is the only choice. However, an ESP security association (SA) can be configured to perform header authentication (AH) or packet compression (IPCOMP) in addition to encryption.

To decide which protocols and algorithms to use we measured the impact of many of the choices on throughput. We found that the combination of encryption (e.g., with 3DES-CBC) and header authentication (e.g., with HMAC-SHA1) reduced the throughput to less than 1 Mbit/s on a Pentium 120 MHz PC. So we decided to only use encryption because we have to support videoconferencing bandwidth of up to 1.6 Mbit/s in each direction. With random keys of 256 bit length, encryption alone provides a sufficient level of packet authentication. For encryption we found that the Blowfish-CBC algorithm is the fastest available with KAME, hence we use Blowfish-CBC with 256 bit random keys.

Configured SAs vs. ISAKMP The IPsec protocols have been designed with the Internet Security Association and Key Management Protocol (ISAKMP) [9] in mind. A security association (SA) is identified by a security parameter index (SPI) and specifies a set of algorithms and keys which two endpoint hosts use to encrypt (or authenticate) their exchange. These parameters are negotiated by the ISAKMP agents on demand, or, alternatively, can be configured statically.

Using ISAKMP can be the only available option, if KAME is to interoperate with certain commercial IPsec implementations, such as Microsoft Windows 2000, Intel PRO/100 PacketProtect, and the Cisco PIX firewall (Cisco's IOS can use static configured SAs.)

The ISAKMP agent for KAME is called 'racoon'. Racoon appears to be one of the less mature components of the KAME suite and it has not been made part of the standard FreeBSD release as of 4.4-RELEASE. We found that in order to compile and use racoon at all, we had to work from a recent KAME snap-kit. Racoon as part of the FreeBSD ports collection was mostly out of date and racoon as part of the KAME snap-kit would require all of that snap-kit. The only alternative to racoon is isakmpd as developed for OpenBSD and although isakmpd can in theory be used on other BSDs and with KAME, we gather that isakmpd is not much more mature.

Since at the beginning of this project, stability and performance of racoon was questionable, we decided to use statically configured SAs initially. Configured SAs are set up typically at time of system boot using the setkey(8) shell command as follows[1]

```
setkey -c <<END
  add $myend $hisend esp $spi_out
    -E $algo $key;

  add $hisend $myend esp $spi_in
    -E $algo $key;
END
```

Each direction from *myend* to *hisend* and vice versa has its own SA with its unique security parameter index (SPI). The SPI is part of the IPsec packet and allows the host to match an incoming packet with its SA. When setting up a multi-way VPN system with configured SAs assigning unique matching SPIs *spi_out* and *spi_in* on each VPN router node is important. If SPIs don't match up, the packets cannot be processed and are lost. In order to be sure that SPIs do match up, we let each end calculate the SPIs using a common formula that combines the overlay network addresses and the role of the endpoint as central (server) or remote (client).

[1] For all code examples, we assume the Bourne shell language sh(1), and we use shell variables as addresses for brevity and readability.

Tunnel vs. transport mode IPsec in tunnel mode is the approach of choice when implementing a VPN for a heterogeneous network. When some machines are not natively IPsec capable, using IPsec in end-to-end transport-mode is not an option. Also, when certain systems may be located behind NAT components, transport mode is likewise not possible, because ESP transport mode encrypts the port numbers which are required for the commonly used port-based NAT.

The normal way to set up an IPsec tunnel only requires defining security policies (SP). If ISAKMP is not used, a pair of SAs is also needed as discussed above. The SPs are set up with the same shell-command setkey(8).

```
setkey -c <<END
  spdadd $mynet $hisnet -P out ipsec
   tunnel/esp/$myend-$hisend/require;

  spdadd $hisnet $mynet -P in ipsec
   tunnel/esp/$hisend-$myend/require;
END
```

The X-Bone [3] and many online tutorials about implementing IPsec tunnels with KAME [10,11], however, suggest implementing IPsec tunnels using IP-in-IP tunneling in combination with IPsec in transport mode. Both approaches result in the same wire format, the difference is only how they are managed [12].

```
ifconfig gif0 $meonhisnet $hisnetmask \
              tunnel $myend $hisend link1

route add $hisnet -interface gif0

setkey -c <<END
  spdadd $myend $hisend -P out ipsec
    transport/esp//require;

  spdadd $hisend $myend -P in ipsec
    transport/esp//require;
END
```

Separating the tunnel configuration from the IPsec processing makes it easier for the user to test and debug the tunnel before enabling IPsec processing where SPIs, protocols, keys, etc. must all match up properly.

Because with IP-in-IP tunnels the VPN overlay network *hisnet* is an entry in the routing tables (even without the explicit 'route add' command shown in the example), packets originating from the tunnel-endpoint and with destination to *hisnet* will be routed into the tunnel. Conversely with only the IPsec tunnel policies, packets originating on the tunnel host will not match the tunnel policy because their source address (*myend*) is not in the space of VPN overlay addresses, but in the space of global IP addresses. Thus one usually needs 4 machines to fully test and debug an IPsec tunnel.

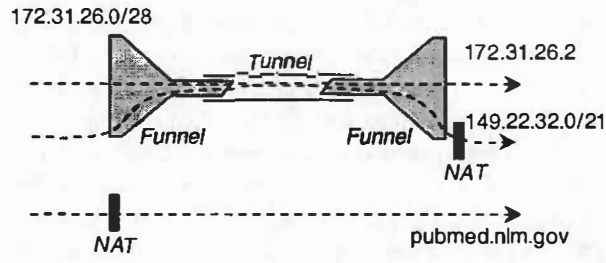

Figure 2: Tunnel remotely links two private subnets. Additional policies act like a funnel routing flows between other networks through the tunnel.

The IPsec tunnel implementation before June 2001 suffered from a severe bug which would cause failures from inbound ESP packets not being properly processed all the way to sudden kernel-panic conditions when more than one pair of tunnels was configured. Fortunately, this bug was fixed by Jun-ichiro (Itojun) Hagino after we could sufficiently isolate the conditions causing this bug to become manifest. This bug affects all FreeBSD releases up to and including 4.3-RELEASE.

Tunnels and Funnels Each physician's home site has two SAs that establish a tunnel to the central nursing home VPN router. The tunnel connects the physician's telemedicine network to the nursing home's telemedicine network. However, the tunnel is not only used for traffic between these two VPN overlays, but for all traffic between the physician's office network and the hospital network and adjacent networks. This provides and extended shell of security wherein other applications, such as the web-based Electronic Medical Record system can be secured as well. Many corporate IPv4 networks today face similar issues, where several IP address spaces exist that need to be included into the VPN without renumbering the entire network.

Thus one tunnel between the central VPN router and each remote site must have multiple ingress and egress rules that "funnel" the packets between the physician's office network and the extended networks through that tunnel.

```
funnel="$wnet1 $wnet2 $rgnet $iunet"

for i in $funnel; do
 setkey -c <<END
  spdadd $mynet $i -P out ipsec
   tunnel/esp/$myend-$hisend/require;

  spdadd $i $mynet -P in ipsec
   tunnel/esp/$hisend-$myend/require;
END
done
```

Corresponding policies need to be set up on the central server in the nursing home. These policies must direct the flows between the extended networks and the physician's office network through the tunnel.

```
for i in $funnel; do
 setkey -c <<END
  spdadd $i $hisnet -P out ipsec
   tunnel/esp/$myend-$hisend/require;

  spdadd $hisnet $i -P in ipsec
   tunnel/esp/$hisend-$myend/require;
END
done
```

Interactions between IPsec, NAT, and firewall
Both IPsec policies and firewall or NAT packet filters are based on the same kind of criteria, i.e., source, destination address protocol and port numbers, etc. In absence of a common framework of filtering rules, it becomes a problem in what order IPsec and firewall filters are executed. No standard exists for the order in which IPsec and packet filters are applied. For KAME IPsec in combination with the IPF firewall/NAT suite the following procedures apply:

a) Incoming packets are first processed by filter and NAT rules and then handed over to the IPsec policies.
b) Outgoing packets are first processed by IPsec and skip outgoing filter or NAT rules.

These procedures are useful because outgoing packets, once processed by IPsec rules, do not require additional NATing or filtering while all incoming packets are first subjected to the firewall rules for intrusion enforcement. This requires enabling passing-rules for the IPsec protocol numbers 50 and 51 for AH and ESP respectively on the inbound leg, but no corresponding rules on the outbound leg. However, with other firewalls, the order may be different and filter rules may be needed for outgoing IPsec or for the incoming packets after IPsec decryption.

3.4 Quality of Service

With coast-to-coast Internet bandwidth in the range of several megabit per second, the need for quality of service arrangements is easy to overlook until one encounters a problem that can only be solved with QoS. In our case we had a bottleneck at the uplink of the remote location, where the cable-modem limits outgoing bandwidth to approximately 100 kbit/s. Such bottleneck can make a H.323-based video-conferencing system unusable. Given that each application loses packets at the bottleneck in proportion of their bandwidth, the more critical but less bandwidth-intensive applications will be impacted more than the highly-redundant video. In our case, audio was delayed by up to 10 seconds and cut out frequently so as to become useless; remote camera control was irresponsive.

The goal of a QoS arrangement under such circumstances is to allocate different bandwidth to the outgoing flows according to the need of the overall application rather than the greed of the individual part. An H.261 video system will try to use as much bandwidth as can be delivered without packet loss; on the other hand, the H.261 codec can cope with lower bandwidths and is automatically adjusted when packet drops are detected. The task of the QoS facility is therefore to drop many packets of the video stream forcing it to lower its output. Conversely, a 64 kbit/s MP3 audio codec will always need those 64 kbit/s of bandwidth to function. TCP-based camera control (typically based on a terminal-server type circuit) will in theory adjust to decreased bandwidth, but not without delays intolerable for a control-feedback circuit. Thus, our QoS arrangement must allocate 64 kbit/s for audio, a small but prompt 9.6 kbit/s for camera control, and must confine the video signal to a budget of approximately 10 kbit/s (discounting other TCP flows that also were given priority over the outgoing video signal).

The only reliable way of discerning the three flows (session control, video and audio) from our video conferencing application (VCON) is by the type of service (ToS) IP-header field.

FreeBSD provides two alternatives to such a QoS scheme, DUMMYNET and ALTQ. DUMMYNET is part of the IPFW suite. Initially invented to simulate realistic network behavior, it can be used for allocating different bandwidth limits and delays for different flows. The three main reason not to use DUMMYNET/IPFW for QoS is if (a) we don't use IPFW but IPF, (b) IPFW had no way of filtering packets by ToS field, and (c) DUMMYNET is too inflexible with bandwidth allocation.

The KAME suite of "Next-Generation-Internet" (NGI) protocols includes Kenjio Cho's Alternate Queuing Framework (ALTQ). ALTQ provides a traffic flow definition language, and several queuing disciplines, including FIFO, priority queue, Weighed Fair Queuing (WfQ) and Class-based Queueing (CBQ). For most applications with a few defined flows, CBQ appears to be the queuing discipline of choice.

After allocating 70 kbit/s for audio, 10 kbit/s for camera control and 15 kbit/s for miscellaneous use (e.g., web-based medical record access), only 5 kbit/s were available for video. This is an impossibly small budget through which one can not expect any reasonable video. In this situation, CBQ allows bandwidth to be borrowed from parent to child classes if the bandwidth available for the parent class is not fully used. This allows excess-bandwidth allocated to audio and unused camera con-

trol bandwidth to be used for video. In practice we can use between 10 and 30 kbit/s for the video channel. This video signal would not be useful for examining patients, but is enough for providing an image of the physician to the patient, which the patient usually appreciates.

Queuing is most effective at the ingress of a bottleneck, in other words on an outbound direction of an interface. However, ALTQ can also measure, mark and drop packets on an inbound direction of an interface. We use this feature to reduce excess load of data into the router, avoiding excess encryption of data that has no chance of being forwarded.

Interaction between ALTQ and IPsec Since any QoS scheme needs to inspect the outbound traffic to discern flow classes, it would be impossible to use QoS effectively on encrypted traffic. Since most bottlenecks are at the ingress of or within public networks, IPsec and QoS could almost never be used were it not for the optional "ECN-friendly" behavior [13] that KAME implements in its IPsec or IP-over-IP tunnels. ECN-friendly means for outbound packets that the tunnel agent copies most ToS bits of the payload packet into the ToS bits of the tunnel IP header. On inbound traffic the congestion notification bit of the tunnel IP header is copied into the ToS bits of the payload packet after decryption (hence the name "Early Congestion Notification", ECN). ECN-friendly behavior is the only way to let ALTQ discern traffic classes in an encrypted tunnel.

Since the ToS field through ECN-friendly behavior is the only way to use ALTQ on an encrypted tunnel, one has to reflect all traffic flows by a specific marking in the ToS bits. This can be done by using the ALTQ on the inbound direction such that characteristics as source and destination address protocol and port numbers can be used to classify the unencrypted traffic. Each packet is then marked with ToS bits indicating its flow class. At the outgoing interface IPsec ESP traffic will be classified only by ToS bits and queuing can proceed as usual on the encrypted traffic.

3.5 Novell NetWare IPX Tunneling

In a corporate networking environment, one frequently has to deal with certain non-IP protocols, most likely IPX. We use FreeBSD as a router to shield the wireless network from excess traffic, and hence we have to route IPX as well as IP. We also, experimentally, support IPX tunneling through the VPN.

FreeBSD includes some support for IPX protocols including multiple Ethernet frame types (Ethernet-II (by default) and 802.2, 802.3 and SNAP through the pseudo-device "ef"), IPX forwarding, and bindary-mode Novell NetWare protocols (file system, printers, login.) IPX routing requires nothing else than enabling the respective IPX forwarding kernel option with sysctl(8) and running the IPX routing agent IPXrouted(8).

For IPX tunneling, we now use Boris Popov's pseudo-device "nwip" (if_nwip.c) that mimics an Ethernet device and encapsulates the packets through UDP/IP packets to a certain tunnel peer. This UDP/IP traffic is then subject to our IPsec tunneling.

As FreeBSD evolves, Boris Popov's nwip code is, however, exceedingly outdated. We had to modify the code to work properly with FreeBSD 4.2-RELEASE, and we could not make it work with 4.4-RELEASE yet. Since this nwip code doesn't appear to be actively maintained any more, we plan to develop a new nwip facility outside the kernel using the generic Ethernet tunneling pseudo-device tap(4).

4 DEVELOPMENT, DEPLOYMENT

FreeBSD comes with a development suite for small stand-alone systems, called "PicoBSD". The FreeBSD installation process itself depends on such small systems to bootstrap a new machine. One can fit a limited system on a single floppy disk, including kernel, programs and configurations (for another PicoBSD adaptation to the SOEKRIS net4501 see http://sourceforge.net/projects/thewall.)

Squeezing a stand-alone operational system on a 1.4 MB floppy disk is possible due to three key functions: (a) the FreeBSD boot loader that can load a compressed image of a memory-resident file system from the diskette and mount it as the root file system; (b) the boot loader that can decompress gzip-compressed a kernel image; and (c) the crunchgen(1) utility that combines multiple programs into one statically linked binary file. This spares many copies of library code that would otherwise be linked repeatedly to each of the programs without the overhead of shared libraries.

The PicoBSD development suite allows the user to generate small BSD system disk images controlled by configuration files in combination with a menu-driven interactive tool src/release/piocbsd/build/build.script. We borrowed the principle approach to developing our system images from the PicoBSD suite, but we have modified the configuration and workflow significantly.

The menu-driven system did not lend itself well to automatically producing several images of the same kind with some minor variations. We therefore replaced the menu-driven system with a straight-forward Makefile approach. Different configurations of images are built by setting different options when make(1) is invoked. The major options are HOST and MODEL. The HOST determines the statically configured host

name, IP addresses, IPX addresses, and VPN tunnel configurations, etc. The MODEL determines on what kind of hardware this host is to be instantiated. Our supported models are (1) the FlyTech NetPC with 8 MB Disk-on-Chip and (2) the SOEKRIS net4501 with 16 MB CompactFlash device.

Conserving Memory Because we have to disable swapping to solid state flash memory, we must reduce memory usage by reducing the kernel to the bare minimum and keeping the memory-resident root file system small. Our root file system uses only 44% of 2 MB compared to the FreeBSD install floppy "mfsroot" that uses 78% of 2.88 MB.

The reduction in size of the root file system is possible because unlike the install floppies, we have a relatively generous 8 or 16 MB flash-ROM available to store most of the program binaries. On the root file system itself, we only use 688 kB for a crunched binary (1822 kB on FreeBSD's mfsroot.) Our root binary only contains init(8), sh(1), mount(8), and fsck(8). When the system boots, init is started that uses sh to execute the autoboot script /etc/rc. This first mounts the flash-ROM disk to /flash and copies a field-editable set of configuration files to /etc. Then a script /etc/rc.real performs a subset of the usual autoboot tasks.

To further reduce memory use, the bulk of the programs are distributed over 3 other crunched binaries. (1) A *setup* crunch file (800 kB) contains only those programs that are needed during initialization (e.g., ifconfig, route, setkey, etc.) (2) A *system* crunch file (900 kB) contains all daemon programs and those programs that run most of the time, and thus, whose code segments would be paged into physical memory anyway. (3) A *user* crunch file (1.9 MB) contains those programs that are only needed when an administrator logs in for remote maintenance, and includes most of the essential UNIX system management and productivity tools (e.g., ps, netstat, ping, telnet, fetch, grep, sed, vi, find, etc.) For each available program the memory-based root file system contains a symbolic link from the usual directories /bin, /sbin, /usr/bin, /usr/sbin, and /usr/libexec to the respective crunched binary.

Robustness and Maintenance The flash file system is mounted in read-only mode so that the router can be powered down without concern to corrupting the boot file system (we don't actually require fsck(8) in the root binary.) The flash file system is only written to at two occasions: saving a field-customized configuration and upgrading the whole system image. For saving customizations applied in the field, the flash file system is remounted writeable and contents of the /etc directory of the running system written to a compressed tar file. This file is reloaded into the running system on rebooting just before the normal /etc/rc autoboot script is executed.

System image upgrade is even simpler: we simply use fetch(1), a command-line HTTP client, to copy the new system image directly to the flash-ROM of the running system, then the system is rebooted. Replacing a mounted flash-ROM file system on-the-fly has been a very reliable process and has caused no problems. Only when the download is aborted incompletely can the system become unusable and requires physical access to recreate. To reduce this small risk in the future, we plan to save a compressed system image on the memory file-system first, before we copy it to flash-ROM. This is the main use where a memory file system with ample free space is invaluable.

Having worked with both kinds of flash-ROM media, Disk-on-Chip and CompactFlash, we find CompactFlash much easier to work with. Being a "consumer product," CompactFlash is much less expensive (generally less than ½ of the Disk-on-chip prices.) But most importantly, the Disk-on-chip devices emulate disk geometry very disadvantageous if not erroneous, causing the boot process to fail on Disk-on-Chip if the compressed kernel and root memory file system image uses blocks above the first 2 MB (approximately) of the disk.

Currently our VPN routers start from a statically configured system image created by the development environment just described. Should configuration parameters change, the system administrator can log into the remote router and manually modify some configuration files and save them back to flash-ROM. Alternatively, changed configurations can be branded on new images and then reinstalled.

The need for manual interaction with the router systems is kept low, and indeed, we have not laid a hand on most of the deployed systems for several months. We have added some preliminary web-based monitoring facility running the simple-http web server that is part of the special PicoBSD source code. We also run a DHCP server (ISC DHCP3) on both the office and private LAN segments so that machines connected to these segments need no manual IP configuration.

5 CURRENT WORK IN PROGRESS

We are currently working some important upgrades to our VPN routers with the objective of making configuration and deployment even simpler and more flexible. These changes include: (1) have outside IP addresses dynamically assigned the ISP's DHCP mechanism, (2) make all devices load their configuration parameters through the network rather than from flash-ROM, and

(3) have all tunnels established dynamically rather than with statically shared keys.

With this redesign of the autoboot process we have removed all configuration parameters that we anticipate being subject to change from the static configuration files (e.g., /etc/rc.conf, /etc/resolv.conf, /etc/dhcpd.conf, and others). A booting system first sends a DHCP request to the outside interface connected to the ISP. When an Internet-link is operational, the system queries all its configuration parameters through DHCP. This includes IP and IPX network addresses and masks, IPsec tunnel and funnel policies, QoS parameters, DHCP server configurations, etc.

In order to query for individualized parameters, each system needs to know its own identity. The only source of identity for a VPN router system is its X.509 certificate and matching private key. Any two system images are exactly the same except for a unique key and certificate file loaded onto the flash-ROM. When the system boots it extracts its own DNS name from the certificate and can then query all configuration parameters as that DNS-name's sub-domains.

The public key infrastructure and certificate is implemented using OpenSSL and kept very simple. Besides the necessary public key information, our certificate profile only include the issuer distinguished name (DN) and a subject alternate name ("general name") of type DNS. Because OpenSSL's "ca" tool does not work for our simple certificates without subject DN (which is a valid certificate as per the PKIX specification [14]), we only use the "req" and "x509" tools and we had to modify parts of OpenSSL. We also added a function into the x509 tool that lets us extract specifically the DNS subject alternate name from a certificate.

In order to set up tunnels dynamically, racoon supports authentication via X.509 certificates and a "passive mode" where security policies are dynamically generated from the client ISAKMP proposal (generate_policy on). Notice the difference between dynamically negotiated security associations (SA) and dynamically established tunnel security policies (SP). Every ISAKMP agent can negotiate SAs, but usually the tunnel SPs still need to be statically configured. However, as we turn to having the ISP assign our VPN routers' outside IP addresses through DHCP, we cannot establish fixed tunnel policies, because that would require knowing the remote tunnel endpoint address.

On the client side, tunnel policies are set up through the DNS based remote configuration. With racoon's *generate_policy* option enabled on the central VPN router, the server's racoon will add a pair of SPs that match the SPs on the client that triggered the initial contact between client and server. However, we found that this alone will not support our funnel policies. The problem is that after the SA is established, ISAKMP will not be involved in any further data exchanges through the tunnel. The client assumes that the tunnel is established and funnels other traffic through the tunnel, however, on the server side no funnel policies exists apart from the one policy that took effect for the first contact. We have modified racoon and added support for funnels. One can now configure racoon to use a certain shell script when an initial contact between tunnel client and server is made. That shell script will then query the DNS-based configurations to determine and establish all the funnel policies. (We borrowed the idea of a shell-escape in a daemon process from the ISC's DHCP client.)

In the future we will use the hardware encryption option based on the HiFn 7951 that is available from SOEKRIS as a mini-PCI module for the net4501 or a full size PCI module for other systems. At a cost of $80 this chip can significantly reduce the CPU cost of encryption and public-key cryptography. At this time, however, this chip is supported on OpenBSD but not yet on FreeBSD.

6 DISCUSSION

Our task was generally to install and configure existing open source components to form one functional and maintainable system, not to develop these components from scratch. Thanks to the work done by others we have been quite successfully in our project. Although we did have to use kernel debugging techniques and modify some components, there were very few problems that others did not fix promptly or that we could not fix ourselves. In this section we list the major issues we found.

6.1 Maturity of the IPsec and KAME

While the KAME IPsec implementation has an excellent track record of interoperability testing [www.vpnc.org] we found several bugs and issues that apparently are manifest only in more advanced and complex scenarios that we had to work with to integrate our system into an existing networking infrastructure. We had network blockage and kernel crashes related with multiple SAs and SPs. We were very pleased with the rapid bug fixes we received once our problems had been isolated. However, our experience suggests that the testing scenarios in interoperability tests may be too simplistic to make those tests valid for "real-world" applications.

Stale SAs, an IPsec/ISAKMP robustness issue. We are not entirely confident yet whether the use of

ISAKMP and racoon in particular will be robust enough. One possible problem case is that the server's and the client's SA data may not be always in synchronization. For instance, when the server is rebooted its SAs are lost, while the client still holds on to the now stale SAs with the server. The client will continue to send IPsec packets to the server using the stale SA, whereas the server does drops those IPsec packets because they don't match any of the server's SAs. The client has no way to notice that the tunnel is broken and the server does not reinitiate an ISAKMP negotiation for a new SA.

This appears as a flaw in the IPsec/ISAKMP protocol design: IPsec with ISAKMP depends on a state (the SA database) that two peers negotiate and must maintain synchronized, but it has no way of promptly recovering from one peer loosing its state. Obviously, the fact that ISAKMP negotiated SAs expires after a certain time (usually several hours) will cause this problem to be solved automatically at time of SA expiration. However, the protocol is still not robust if network downtimes can happen, even if only for several minutes. Also, the ISAKMP negotiation is too costly both in terms of elapsed time and CPU time to expire SAs very frequently. We believe that the IPsec/ISAKMP protocol needs to be amended to provide for detecting and resolving the problem of stale SAs promptly.

6.2 Excessive IP Packet Matching

Most network components such as (1) router, (2) packet filter, (3) NAT, (4) IPsec policy engine, (5) IPsec association matcher, (6) traffic shaper, (7) traffic conditioner, (8) raw IP handler, and (9) IP-in-IP tunnel handler all use rules that operate on some characteristics of IP packets, typically of the IP header. Even though the task of matching packets with criteria is common to all of these components, each uses its own syntax for the criteria and/or its own code to compare actual packets against those criteria. Not only will users have some added difficulty mastering the many criteria languages, the duplication in poorly optimized (linear search!) matching code contribute to "kernel bloat" and increase the CPU time spent on each packet flowing through the system. We feel that the packet processing components and their configuration could be better coordinated perhaps using a more efficient packet matching algorithm such as the Berkeley Packet Filter (BPF) [15].

6.3 Automation-Friendly Tools

UNIX is well-known for its large set of tools that "do one thing and do it well" [McIlroy in 16]. Together with pipes and the shell command interpreter, one can build powerful automated scripts quickly. We do this extensively creating most of the networking components' configuration files from templates and generators at system boot time. In a few cases we saw no choice but to create some of our own tools.

IP address calculations. We missed a tool that would allow us to do some common address calculations on the fly. This was necessary because we wanted to reduce the number of manually maintained configuration parameters but the various configuration files required many parameters that could be derived from fewer parameters. For example, some components (e.g., dhcpd(8)) require netmasks (e.g., 255.255.255.0) for specifying subnets while others (e.g., setkey(8)) would take only network prefix lengths (e.g., /24). So we had to have a tool that could convert between both. Some would require a host-independent network address (e.g., setkey(8) or route(8)) that could be derived from an IP address in the network (e.g., 172.31.24.1/24 has network address 172.31.24.0). For these and other address calculations we developed *ipac*, am IP Address Calculator. The ipac tool can increase or decrease host numbers, or network numbers, test if one network subsumes another network and convert addresses in different representations (hexadecimal, dotted decimal, simple decimal). This tool simplified many of our configurations and we believe it could be useful for a broader community. (The only shortcoming being that it only supports IPv4 addresses at this time.)

Querying the DNS. While there are many tools to query the DNS, such as nslookup or dnsquery, all of the tools we found seemed to be geared to interactive use or at least human interpretation of their output. For our DNS-based remote configuration mechanism, we required a tool that could query the DNS for various resource record types (e.g, A, TXT, PTR) and return the resulting data in a simple non-verbose form. Thus we developed *dnsq*, a DNS Query tool that is based on the standard BSD resolver and uses Dave Shield's resparse library for parsing the DNS results.

7 CONCLUSIONS

This work has shown that FreeBSD is a very useful platform to develop custom networking solutions that offer at a very low cost a host of functionality that we could not otherwise implement using commercially available products even for prices 5 to 10 times higher. Self-designed network infrastructure equipment such as ours can be economically feasible even for such institutional users who do not consider themselves hardware systems developers. The effort it takes to acquire skills in designing custom FreeBSD-based solutions probably is not much higher than what it takes to successfully

implement a commercial product: instead of wading through amounts of sales brochures hunting for the information to figure out if a commercial product will meet one's needs, instead of having endless phone calls to figure out the pricing options with sales representatives who cannot answer technical questions, instead of spending hours on waiting loops with customer service call centers to figure out that the commercial product has one little missing feature or bug that will fail the entire implementation or require awkward workarounds; that same time can be more productively spent on developing a custom solution with standard UNIX-based tools that can be made to work even if it does not initially work out of the box. The open source community around FreeBSD and KAME have provided excellent "customer support" to us: we received one critical kernel patches within a month and another one in just 24 hours, few questions remained unanswered on e-mail lists, and those that didn't get answered could eventually be answered by studying the source code.

We plan to contribute our development environment and tools back to the open source community once we have finalized our upcoming major version. The code will be available from http://aurora.regenstrief.org.

ACKNOWLEDGEMENTS

This work was performed at the Regenstrief Institute for Health Care and supported by the National Library of Medicine contract NLM-99-106/RMC.

REFERENCES

1 Weiner M, Schadow G, Lindbergh D, Warvel J, Abernathy G, Dexter P, McDonald CJ. Secure Internet videoconferencing for assessing acute medical problems in a nursing home facility. Proc AMIA Symp 2001. pp. 751-756.

2 International Telecommunication Union. Video codec for audiovisual services at p × 64 kbit/s [Recommendation H.261]. Geneva, 1990. The Union.

3 Touch J. Dynamic Internet overlay deployment and management using the X-Bone. In Proceedings of the 8th International Conference on Network Protocols, Osaka, Japan, November 14-17, 2000. pp. 56-68. Available from: http://www.isi.edu/touch/pubs/icnp2000.

4 Rekhter Y, Moskowitz G, Karrenberg D, de Groot GJ, Lear E. Address allocation for private Internets [RFC 1918]. Network Working Group, 1996.

5 Conoboy B, Fichtner E. IP filter based firewalls howto. Available from: http://www.obfuscation.org/ipf.

6 Kent S, Atkinson R. Security architecture for the Internet protocol [RFC 2401]. Network Working Group, 1998.

7 Kent S, Atkinson R. IP encapsulating security payload [RFC 2406]. Network Working Group, 1998.

8 Kent S, Atkinson R. IP authentication header [RFC 2402]. Network Working Group, 1998.

9 Maughan D, Schertler M, Schneider M, Turner J. Internet security association and key management protocol (ISAKMP) [RFC 2408]. Network Working Group, 1998.

10 Rushford JJ. Setting up a FreeBSD IPsec tunnel. FreeBSD Diary, June 2001. Available from: http://www.freebsddiary.org/ipsec-tunnel.php

11 Tiefenbach J. FreeBSD IPsec mini-howto. DaemonNews, 2001 Jan. Available from: http://www.daemonnews.org/200101/ipsec-howto.html

12 Touch J, Eggert L. Use of IPSEC transport mode for virtual private networks [Internet draft-touch-ipsec-vpn]. Nov 24, 2000.

13 Floyd S, Black DL, Ramakrishnan KK. IPsec interaction with ECN [Internet draft draft-ietf-ipsec-ecn-02.txt]. December 1999.

14 Housley R, Ford W, Polk W, Solo D. Internet X.509 public key infrastructure certificate and CRL profile [RFC 2459]. Network Working Group, 1999.

15 McCanne S, Jacobson V. The BSD packet filter: a new architecture for user-level packet capture. In Proceedings of the USENIX Winter Conference, San Diego, CA, January 25-29, 1993. pp. 259-270.

16 Salus PH. A quarter-century of Unix. Addison Wesley 1994.

SystemStarter and the Mac OS X Startup Process

Wilfredo Sánchez
wsanchez@mit.edu

Kevin Van Vechten
kevinvv@uclink4.berkeley.edu

Abstract

This paper documents a program in Mac OS X called SystemStarter. SystemStarter brings the system from its initial state up to a state where basic services are running and a user may log in. It replaces the previous `/etc/rc` startup sequence employed by Mac OS X's predecessors in order to address some additional goals set forth by the Mac OS X project.

SystemStarter is part of the BSD subsystem in Mac OS X, now known as Darwin, though it's creation predates Darwin as an open source project; it was therefore created by a single author, though it now enjoys several contributors.

Background

Mac OS X is the latest version of Apple's operating system for its Macintosh computer line. One of the unique attributes of Mac OS X as compared to previous versions of Mac OS is the use of BSD as its base system implementation. During the Mac OS X operating system bringup, a great deal of effort was spent on integrating the many facilities of Mac OS on this new BSD foundation [1]. For the most part, BSD facilities did not have to change significantly in order to accomplish this. One of the few that did is the system startup sequence.

Mac OS X started largely form the code base for a prior OS product called Mac OS X Server, once better known by its code name "Rhapsody." Rhapsody uses a startup sequence inherited from OpenStep. The kernel launches init, which runs a script `/etc/rc` as per BSD convention. `/etc/rc` would then run scripts in the directory `/etc/startup` in lexical order based on filenames. The script `0100_VirtualMemory` would run before the script `1600_Sendmail`, for example. When these scripts finish, `/etc/rc` would exit and init would then bring up the multiuser login prompt as configured in `/etc/ttys`.

The Problem

This mechanism was simple, and by allowing separate scripts for each service to be run at startup, it allowed users to insert services into the startup sequence in a straightforward manner. However, there were several drawbacks:

• The lexigraphic ordering is fragile. If we were to change the order in which scripts run, or insert or delete a new standard service into the sequence, we may have to renumber several of the files. This means that a user (or a vendor package installer) would have to place additional scripts into different locations in the order depending on the system version, and it would be impossible to know that the future system releases would not again break the ordering.

• The startup sequence is inherently limited to serialization by this design. It may make a lot of sense to run a disk-intensive task such as cleaning up `/tmp` while simultaneously running a network-blocked operation such as requesting a DHCP lease. In the `/etc/startup` scheme, these were always run in sequence, failing to take advantage of the systems ability to manage multiple resources simultaneously.

• It was not easy to know which scripts were installed by the user, and which are part fo the system software, as they are co-mingled.

• /etc is hidden by the Finder. A user should be able to copy and manage startup scripts at a well-known location, but /etc, /var, /usr, and other Unix directories are by default hidden from view.

• Graphical startup was difficult. The window system must start very early in order to display startup progress, and the startup scripts would need some way to update the display. Rhapsody used a command-line tool which could draw directly to the display's frame buffer before the window system was running, but this was difficult to maintain and highly limiting. You could not, for example, localize text, as font support wasn't available, etc.

Alternatives

Other BSD systems were using schemes that we similar, usually involving a couple of scripts in addition to /etc/rc which were specific to networking, loading kernel modules, etc.

System V systems (such as Solaris, and in it's own quirky way Linux) had a fairly different mechanism that was worth studying.

System V uses separate scripts similar to our /etc/startup scripts, but each script has the ability to stop as well as start a service, which provides for the possibility of a shutdown sequence and the ability to start and stop services after the system is running as the user needs them [2]. Extending /etc/startup to allow this would be fairly easy, and the value of doing so was clear, so it quickly made it to the list of features to include in a new scheme.

Another feature in System V startup is the concept of runlevels, which represent different system states [3]. Single user mode is one runlevel, multiuser mode is another. There are other runlevels for intermediate states, as well as pseudo-runlevels for shutdown and restart. This concept didn't last long on the feature list. The value of having runlevels is questionable at best, their semantic meaning is vague (which is not helped by their being labeled with numbers instead of names), they add a significant degree of complexity in managing startup scripts (Which runlevels provide which services? How do you handle transitions?), and explaining all of this to a non-technical user (in fact, even a technical user) is not a simple exercise.

It should be noted that NetBSD began work on a replacement for its /etc/rc [4] shortly after work on SystemStarter had begun. Wilfredo Sánchez, who had started the work on SystemStarter, passed along his current design ideas, but though the goals overlapped, the projects had already gone down different design paths and not much collaboration came of it.

Solutions

The initial design was to create a new program, called SystemStarter, which will be responsible for managing the startup process. Services will be described by "startup items," which know how to start a service, the service's dependencies, and other information such as user-visible strings. SystemStarter will compute an order for starting items based on the dependencies and see that the items are run accordingly.

SystemStarter is implemented in C using Mac OS's CoreFoundation framework, which provides a number of useful data types (strings with encoding support, arrays, hash tables) and functionality (XML parsing, interprocess communication, run loop, object reference counting) in as object-oriented a fashion as allowed by the C language.

Startup Items

Startup items are comprised of several parts, and implemented as a directory containing several well-defined files. Each file addresses a specific part of the item's functionality. In Mac OS parlance, this is termed a "bundle" directory. In the application toolkits, bundles are often treated as single objects, similar to files.

As with other startup mechanisms, a program, usually a shell script, describes the activity required in order to start the service(s) provided by the item. The script is given a "start" argument during system startup. In the future, the "stop" and "restart" arguments will be sent to the script at other times. (See "Ongoing Work" below). /etc/rc.common provides some useful functions (for example, the GetPID function used here gets a process ID registered in /var/run, if available) as well as a standard mechanism for handling arguments passed in by SystemStarter. For example, a script for starting cron might look like:

```
#!/bin/sh

. /etc/rc.common

case $1 in
 start)
   ConsoleMessage "Starting cron"
   cron
   ;;

 stop)
   if pid=$(GetPID cron); then
       ConsoleMessage "Stopping cron"
       kill -TERM "${pid}"
   else
       echo "cron is not running."
   fi
   ;;

 restart)
   ConsoleMessage "Restarting cron"
   if pid=$(GetPID cron); then
       kill -HUP "${pid}"
   else
       cron
   fi
   ;;

 *)
   echo "$0: unknown argument: $1"
   ;;
esac
```

In addition to a procedure for starting its service(s), a startup item needs to provide enough information so that it can calculate an order for all of the startup items. This information is described in a property list file in the item. The format of the file is either a NeXT property list, or XML. (XML is the preferred format for preference files in Mac OS X, and can be edited with a tool like PropertyListEditor, but we will use the NeXT format here because it is significantly more compact and readable, and friendly to manipulation as raw text.) And example for `cron`:

```
{
 Description     = "timed execution services";
 Provides        = ("Cron");
 Requires        = ();
 Uses            = ("Cleanup", "Network Time");
 OrderPreference = "Late";
 Messages =
 {
   start = "Starting timed execution services";
   stop  = "Stopping timed execution services";
 };
}
```

A human-readable description of the item is presently used in debugging only, but may also be used by a startup manager application in the future. Strings are also provided for display to the user when the item is starting and stopping. In the future, these strings will be passed by the script via IPC to SystemStarter, rather than hard-coded in the property list, which will allow for more accurate progress indicators.

A list of services provided by the item is noted, for use by dependent items. For modularity, it is generally best that separate services each reside in separate items, but there may be cases where a single server is used to provide more than one service, therefore a list is allowed. Similarly, a list of services on which the item is dependent is given. These fall into two classes. Some services may be required, in which case, the SystemStarter will not attempt to start the item unless the requirements are started first (which may mean the item never starts). Other services are desired, but optional. For example, cron is a time-based service, which makes it desirable that the computer's clock be synchronized with the network before starting cron, but failure to do so should not prevent cron from starting at all. In this example, cron also wishes to wait on system cleanup, and has no hard requirements.

CoreFoundation's property list parser is used to read the property list and generates an object graph using its CFDictionary, CFArray, and CFString object types. The items are represented in memory as CFDictionary objects. The original plan was to generate a new object graph directly representing the dependency tree for fast searching of dependents. However, in the current implementation items are simply stored in a CFArray and searched linearly. Whenever an item's script exits, all remaining items are checked to see if their dependencies are now met in light of the newly available service(s). The code to manage an array is much simpler, and given that the number of startup items on a system is not expected to be large, the anticipated performance benefit of generating a dependency tree in memory is negligible.

Additional files in the startup item contain localized versions of the strings provided for user-visible display.

Filesystem Layout

Traditionally System V [5], Linux [6], and BSD systems search for startup scripts in a subdirectory of `/etc`, such as `/etc/init.d`. This is undesirable for SystemStarter because the /etc directory is hidden from view in Mac OS X, and because it does not allow third-party startup items to be easily distinguished from the standard items provided by the system distribution. In order to address both of these issues, SystemStarter searches for startup items in one of several directories. Startup items provided with the system distribution are stored in the `/System/Library/StartupItems` directory; these are said to be in the "system domain." Users and third-parties are encouraged to place items in the `/Library/StartupItems` directory, known as the "local domain." Items in the local domain are not deleted or replaced when the system is upgraded. Furthermore, if an item in the local domain provides the same service as an item in the system domain, the item in the local domain takes precedence. In this way, a user may supersede a standard item without worrying about the behavior reverting after a system upgrade.

Work is in the design stage to provide support for a "network domain" which could be used to ease remote administration of many machines. After mounting a site-local NFS filesystem, SystemStarter could be prompted to search for startup items in `/Network/Library/StartupItems`. The network domain would take precedence over the system domain, but not the local domain.

Graphical Startup

The majority of Unix-variant systems boot in a text console and print out significant debugging information as the system starts up. While this is useful at times, it can be rather baffling to most consumer users. In Rhapsody, we had a program called fbshow, which could draw to the display's frame buffer directly during startup, after which we would start the (PostScript) window server. It would draw a progress panel on screen and could print status text in one font. This was inflexible in that the graphics it used were compiled into the binary, and text was not internationalizable (eg. no Japanese fonts). In Mac OS X, the window server was far more lightweight, and could be started very early in the startup process. This gave full font support, plus all of the display features of CoreGraphics (a.k.a. Quartz), such as PDF rendering and compositing.

It should be noted that because SystemStarter boots the system, its failure due to a crash can be catastrophic to the user. The more API the program draws on, the more libraries need to be loaded, and the greater likelihood of failure due to something like a corrupt file on disk. This is particularly relevant to SystemStarter, because *it* will be responsible for running fsck, which checks for corrupt files and repairs them. For this

reason, rather than making the SystemStarter dependent on CoreGraphics framework (and whatever CoreGraphics depends on), the built-in display functionality in SystemStarter is text based and the Quartz code is placed in a loadable module. If the module fails to load, SystemStarter falls back to text-mode. This also enables additional modules to be written, which proved useful in Darwin, where an X11 module can provide graphical boot using XFree86.

Ongoing Work

Written by Wilfredo Sánchez as an employee of Apple, SystemStarter was first publicly released in Darwin 1.0 and remained mostly unchanged through the release of Mac OS X 10.0 (which corresponded to the Darwin 1.3 release). It successfully provided a user-extensible system startup mechanism, and a graphical startup consistent with the Mac OS experience. After the release of Mac OS X 10.0, Darwin Committers Wilfredo Sánchez and Kevin Van Vechten continued work on SystemStarter to provide a more complete feature set.

Starting and Stopping Services

System V and other BSD systems provide a mechanism to start and stop services after the boot sequence. This feature is valuable, and useful to the system control panels, which allow users to enable and disable services as they see fit. However, care should be taken to ensure that when services are started or stopped, dependencies are accounted for. Work is being done on SystemStarter so that an item's dependencies are accounted for automatically using the dependency graph of each item.

For example, if the user asks to enable NFS, SystemStarter should ensure that portmap is running. Similarly, if portmap is terminated, SystemStarter should ensure that NFS is shut down as well. This is a difficult exercise, as the interaction between the control panels, the service manager, and the user can become rather complicated. For example, the user should be informed that shutting down one service will result in others shutting down as well. This information is tracked by SystemStarter, but would somehow have to get to the user via the control panel. Additional work is still in the design stage which may add a way for applications like the control panels to pass useful information like this from SystemStarter to the user.

Parallel Startup

One of the goals of SystemStarter going forward is to transition from a serialized startup sequence to a more flexible parallel startup sequence. A side effect of having a serialized startup sequence is that only one service can be brought up at a time. This limitation prevents the startup sequence from taking any advantages of the operating systems ability to schedule multiple tasks simultaneously such that system resources are maximally used. For example, if the system was booted after a power failure or crash, the filesystems on disk will be dirty and must be verified before they are mounted. This process is very disk intensive and can take some time. Similarly, many systems are configured to acquire their network parameters via a service like DHCP or NetInfo. When the network is busy, this can also take some time while waiting on a response from the network service. If startup is serialized, the system must wait on the disks to be checked and then wait on the network configuration. However, there is no reason why both of these cannot happen at in parallel, so that the actual time spent during startup is only that of the longer of the two services, rather than the combined total of both.

Because SystemStarter uses a dependency graph for startup items rather than an ordered list, it is possible to implement a sequence where items run in parallel. As new services become available, each pending service can be started if its dependencies are then satisfied.

Partial Startup and Shutdown

There are times when administrators wish to boot into single user mode to debug or recover parts of the system. In those cases, it is often desirable to bring the system up to an arbitrary point in the startup sequence. While the System V approach offers runlevels to allow the system to be in one of several predefined states, the runlevels are discrete and offer no assistance when a state other than one defined by one of the runlevels is desired.

Work is underway to enable SystemStarter to allow the system administrator bring up enough of the system to provide a specified service. For example, requesting that the "`Network`" service be brought up will start up all services required to use the network, but will not start any additional services. SystemStarter uses its dependency graph to determine what the specified service's prerequisites are. SystemStarter then selects only the items which are required, and performs the standard startup sequence using this subset of startup

items.

It is also possible to take advantage of the dependency graph to providing a logical system shutdown sequence. Traversing the dependency graph in the opposite direction from startup, SystemStarter runs each script with a "`stop`" argument such that each script's dependents are stopped before the antecedent is stopped. For example, a request to stop the "`Network`" service will stop all of the services that require the network, then bring down the system's network interfaces.

Interprocess Communication

During a traditional startup sequence, startup scripts print messages to the console and system log to report their progress for logging and debugging purposes. However, during graphical boot, it is desirable to print messages which indicate to the user (at a simpler level) what is going on. An inter-process communication mechanism in being implemented so startup scripts can send messages to SystemStarter which in turn displays them during graphical boot.

During the startup sequence, SystemStarter listens for messages on a named Mach port. Messages are composed of an XML property list which specifies the type of message and its arguments. Startup scripts use a provided tool to send a message to SystemStarter which will be displayed on-screen. Each message contains a token identifying the startup item which originated it. SystemStarter then attempts to localize the message based on the localization dictionaries provided with each startup item. Finally the localized message (or the original message if no localization could be provided) is displayed.

The XML property list format for IPC messages was chosen to ensure maximum forward and backward compatibility between future versions of SystemStarter and tools that communicate with it. Mach ports were chosen for the IPC mechanism because Darwin's CoreFoundation library, which SystemStarter already used extensively to manage its internal data structures, provides a simple API for sending messages between processes using Mach ports. Additionally, Mach port invalidation callbacks are used to monitor startup item termination, allowing both process termination events and IPC events to be handled from a single event loop.

More IPC message types are planned. Specifically, messages allowing startup scripts to report the success or failure of a particular task would be useful, as items may provide multiple services and accounting for which specific services are running would improve SystemStarter's dependency tracking. Ultimately startup scripts may be required to respond to a "`status`" command, reporting to SystemStarter the status of each service the item provides. This information will be useful in determining which services the system can provide at a given time, as well as what state graphical controls should be set to when displaying system control panels.

Conclusion

In its initial release, SystemStarter succeeded in removing the fragile lexigraphic ordering of startup items, providing a graphical boot sequence, and separating third-party scripts from those of the system distribution. Although SystemStarter differs significantly from other system startup mechanisms, the startup item scripts are fairly similar to those found on other systems. Porting startup items to Darwin involves slight modifications to a startup script and the creation of a property list file describing the item. Thus, SystemStarter effectively accomplished its goals with low overhead and a minimal loss of compatibility.

Availability

SystemStarter is available in Mac OS X and Darwin systems. The source code of SystemStarter is available as part of the Darwin project at:
http://www.opensource.apple.com/

About the Presenters

Wilfredo Sánchez is a 1995 graduate of the Massachusetts Institute of Technology, after which he co-founded an Internet publishing company, Agora Technology Group, in Cambridge, Massachusetts; he then worked on enabling electronic commerce and dynamic applications via the world wide web at Disney Online in North Hollywood, California. Fred later worked as a senior software engineer at Apple Computer in Cupertino, California, primarily on Darwin, the BSD subsystem in Mac OS X, as a member of the Core Operating System group, and as engineering lead for Apple's open source projects. He continues to work on Darwin as a volunteer developer. Fred is also a member of the Apache Software Foundation, and a contributor to various other projects, including NetBSD and FreeBSD. He now works at

KnowNow, Inc. as developer community manager.

Kevin Van Vechten is an undergraduate at the University of California, Berkeley, majoring in Electrical Engineering and Computer Science. He works as a consultant specializing in custom database and networking solutions. Kevin also contributes to Darwin and other various projects.

References

[1] Wilfredo Sánchez; *The Challenges of Integrating the Unix and Mac OS Environments*; USENIX 2000 Technical Conference; San Diego, California; 2000

[2] Sun Microsystems, Inc.; *How to Use a Run Control Script to Stop or Start a Service*; System Administration Guide. Volume 1. Solaris 8 System Administrator Collection. Fatbrain, February 2000. 116.

[3] Sun Microsystems, Inc.; *Run Levels*; System Administration Guide. Volume 1. Solaris 8 System Administrator Collection. Fatbrain, February 2000. 109-110.

[4] Luke Mewburn; *The Design and Implementation of the NetBSD rc.d System*; USENIX 2001 Technical Conference; Boston, Massachusetts; 2001

[5] Sun Microsystems, Inc.; *Adding a Run Control Script*; System Administration Guide. Volume 1. Solaris 8 System Administrator Collection. Fatbrain, February 2000. 117.

[6] Red Hat, Inc.; *Behind the Scenes of the Boot Process*; The Official Red Hat Linux Reference Guide. Red Hat Linux 7.2. Red Hat, Inc., 2001.

Log Monitors in BSD UNIX

Brett Glass, *Glassware*
P.O. Box 1693
Laramie, WY 82073-1693
http://www.brettglass.com/mailbrett.html
BSDCon 2002 slides at http://www.brettglass.com/logmonitors/

Abstract

A log monitor is a process, or daemon, which monitors log messages produced by a computer system and the programs which run on it. A properly designed log monitor can recognize unusual activity (or inactivity), alert an administrator to problems, gather statistics about system activity, and/or take automatic action to contain a threat. It can even "learn," over time, what is normal and identify message traffic that may betray an abnormal situation. Languages which facilitate string processing and pattern matching, such as Perl and SNOBOL4, are good choices for the implementation of log monitors for this reason. Included in this paper is source code for a log monitor that identifies and blocks attacks from Code Red, sadmind/IIS, Nimda, and similar worms. Policy issues -- including the usefulness of "amnesty" to prevent inadvertent blocking of innocent third parties -- are discussed. The work described in this paper --which is still progressing at this writing -- will hopefully culminate in the release of a general purpose log monitoring facility.

[Note: Because the camera-ready copy of this paper was submitted two months prior to the BSDCon 2002 conference to allow time for printing, and was required to conform to strict size limits, the version which appears online may contain more detail as well as new information gleaned from ongoing work. For the latest version, and/or to follow up the many references via HTML links, access the master copy at http://www.brettglass.com/logmonitors/paper.html on the World Wide Web.]

1 Introduction

System administrators' time is valuable. Few, if any, can afford to spend many hours each day poring over the voluminous log files generated by systems and network applications. Yet, if an administrator fails to recognize quickly and respond to anomalous events chronicled in log messages, systems or entire networks can be abused, "hijacked" for use in illicit activities, and/or removed from service by malicious parties. Log monitors aid the administrator by automatically filtering and digesting the flood of log information -- and, in some cases, responding on his or her behalf.

2 What is a Log Monitor?

A *log monitor* is an agent which responds automatically to conditions revealed by one or more system log messages. The response may consist of autonomous action to handle a situation and/or the notification of a human administrator.

A *stateful log monitor* is a log monitor that infers the presence of a condition requiring attention by compiling data from more than one log message. It may simply note the number and/or frequency of log messages related to a particular type of activity or may generate more sophisticated cumulative statistics from those messages.

2.1 Capabilities of Log Monitors

What can a log monitor do? Among other things, it can:

- Detect abnormal usage patterns
- Recognize system or network abuse (e.g. spamming and mail bombing)

- Catch worms and other malware
- Detect vulnerability scans (e.g. port scans)
- Detect intruders (or attempted intrusions)
- Detect resource shortages (e.g. slow response times, out-of-memory conditions, out-of-disk conditions, inadequate swap space)
- Detect imminent or actual system failures
- Compile statistics in real time (including running averages, etc.)
- React to conditions by notifying an administrator and/or taking immediate action

While it is useful for a log monitor to be able to recognize failures or threats for which it has been given a human-crafted "signature," it can also incorporate more subtle heuristics. By accumulating statistics about what constitutes "normal" activity, log monitors may be able to recognize anomalous behaviors that a human system administrator might at first overlook, such as the cessation of events which normally happen with a given frequency.

2.2 Examples of Existing Log Monitors

`swatch` ("Simple Watchdog") [1], developed by Stephen Hansen and Todd Atkins of Stanford University, and `2swatch` [2], an enhanced version of `swatch` developed by the Pacific Institute for Computer Security (PICS) research group at the San Diego Supercomputer Center, are two very simple general-purpose log monitors. `swatch` accepts a configuration file whose entries consist of patterns and lists of actions. (A sample pattern/action block is shown in Listing 1.)

```
watchfor /ANONYMOUS FTP LOGIN REFUSED FROM (\S*)/
 mail addresses=admin,subject=Attempted anonymous FTP from $1
 exec blackhole $1
```

Listing 1: Sample pattern/action block for swatch

It then "tails" a system log file looking for the regular expression specified in the "watchfor" line of the block. If the expression is matched, swatch takes the series of actions that follow. Possible actions include producing a beep, writing a message in a specified color to the console, sending an e-mail message including the log entry that was matched, or executing an arbitrary command. (In the example above, the $1 causes the text matched by the subexpression within the first set of parentheses to be interpolated.) `swatch` incorporates features to allow throttling and to recognize patterns only during certain hours of the day and/or days of the week.

`2swatch` extends `swatch` by offering the possibility of deferred as well as immediate action. `2swatch` can accumulate a series of messages matching a pattern into a report that is e-mailed as a single message. This prevents mail systems from being flooded with e-mail messages that each contain a single log entry. Unfortunately, neither `swatch` nor `2swatch` can perform stateful monitoring beyond their respective throttling and accumulation functions, though it is of course possible to implement statefulness via cleverly designed external programs or scripts. Nor can these two programs, by themselves, correlate entries from more than one log. A final drawback of these two programs is that they have been released under licensing terms (the GPL and a unique "no commercial use" license, respectively) which hamper or prohibit their reuse in commercial products.

The primary objective of the work described in this paper is to develop a generalized and portable framework which eases the creation of customized log monitors and overcomes the limitations described above. It is intended that the results of this work, which is initially being performed on FreeBSD, be released under "truly free" (i.e. MIT- or BSD-style) licensing terms so as to permit adaptation to other operating systems and commercial as well as non-commercial reuse.

2.3 Log Monitors vs. Log Analyzers

A *log analyzer* differs from a log monitor in that it does not operate in real time (or nearly real time) but is run against system logs after the fact. Many such utilities exist, especially for Web and mail servers. At this writing, log analyzers are more common than log monitors. Most implementations of BSD UNIX come with primitive but helpful log analysis scripts that report significant log events to an administrator via e-mail. These scripts are often found at `/etc/daily`, `/etc/weekly`, and `/etc/monthly` and are usually run at appropriate intervals by the `cron(8)` [3] daemon. (In recent versions of FreeBSD, the default `/etc/crontab` file instead activates Trainer and Somers' `periodic(8)` [4] utility, which in turn runs these scripts from the `/etc/periodic` directory.) Apache's `logresolve` program performs a simple analysis of Apache log files that consists primarily of efficient reverse domain name resolution. Tom Boutell's `Wusage` [5] web log analysis utility, released as Shareware, is a much more sophisticated log analyzer for Apache; it can generate daily, weekly, monthly, and annual reports.

Some log analyzers are designed to "wake up" periodically and scan the latest entries in a system log, acting, if appropriate, upon what they "see." (Kai's Spamshield [6] , which detects incoming spam and blocks the sender via a "blackhole" route, is an example.) If the interval between scans is sufficiently short, such log analyzers can perform some of the functions of a log monitor, detecting and handling conditions which require timely but not instantaneous attention.

3 Log Monitors and BSD syslogd

In BSD, the kernel and most daemons traditionally log their activities via `syslogd`, the system logging daemon, originally written by Eric Allman. Messages from the kernel are received via the pseudo-device `/dev/klog`, while messages from applications are received via the UNIX domain socket `/var/run/log` or via UDP socket 514. In the early, more trusting days of the Internet, `syslogd` listened on this socket and by default accepted any message that came in. But in most modern UNIX implementations it does not do this, because -- as stated in many versions of the `syslogd` manual pages -- it was "equivalent to an unauthenticated remote disk-filling service." [7]

3.1 syslogd Facilities, Priorities, and Tags

The standard Berkeley `syslogd(8)` [8] is configured via the file `/etc/syslog.conf`, which specifies the disposition of log messages. In traditional Berkeley UNIX, the messages are sorted according to a *facility* code and a *priority* or *severity level.* These parameters are used to route each log message to one or more destinations, including the system console, the terminals of specific users, other machines, files, and/or programs. Modern UNIX and UNIX-like systems have many more facilities than were contemplated in the original UNIX logging scheme. Nonetheless, most implementations have hewn to the traditional list of facilities defined in 4.4BSD for the sake of compatibility and tradition. A few (such as FreeBSD) have added additional facility codes, including **`console`**, **`ntp`**, and **`security`**, and DEC ULTRIX uses facility number 10 (**`authpriv`** in many versions of BSD) to log events for its AdvFS filesystem. [9] Unfortunately, these additions and conflicts may hinder software portability. What's more, the facilities which were added frequently do not reflect a consistent design philosophy. It is unclear, for example, why the authors of FreeBSD's `syslogd` implementation felt that NTP was deserving of its own facility code while DNS, DHCP, PPP, and HTTP (or, for that matter, any of the many other protocols listed in `/etc/services`) were not.

Further confusing the issue of how to classify log messages is the fact that many recent versions of `syslogd` have added the ability to sort messages via strings called "tags," which are transmitted to the logging daemon along with each message. Tags usually (but not always) contain the name of the program that generated the message. (At this writing, the ability to sort messages by tag is available in OpenBSD and FreeBSD but not in NetBSD.) To facilitate logging across networks, some implementations of `syslogd` add yet

another sorting criterion: they allow messages to be dispatched according to the name of the host from which they originated.

As if all of this weren't enough, there's yet another fly in the ointment. While Berkeley `syslogd` itself is able to sort messages by facility, severity, tag, and originating host, it does not record the facility and severity level in each log message, and in most implementations there is no way to make it do so. This makes it difficult for a log monitor (and, in some cases, for humans) to use these same criteria to sort messages that syslogd has aggregated into a single log file. The version of `syslogd` found in recent versions of FreeBSD is one of the few that provides a solution to this problem. If the daemon is started with the -v ("verbose") command line option, the facility and priority level are logged as numbers. Specifying -v twice causes them to be shown by name between angle brackets (e.g. `<auth.notice>`). The latter choice, while it consumes a bit more disk space, makes the logs much easier both to read and to monitor. It would be a relatively simple matter to add similar capabilities to other `syslogd` implementations.

3.2 Monitoring Techniques for Use With syslogd

A log monitor can monitor the output of `syslogd` in one of three ways. The most common technique is to "tail" the log, either by accepting the output of the "tail -f" as input or by incorporating a module that provides equivalent functionality (e.g. Matija Grabnar's File::Tail [10] for Perl). The downsides of this technique are twofold. First, the application (or the `tail(1)` utility) must "poll" the file at regular intervals to see if it has grown. This consumes resources even if there is nothing to be done. Secondly, if the log file is "rotated" (most often done by renaming it and creating a new file for future input), the application may be left "watching" the old file (which is no longer growing) and miss all subsequent messages. (This is a problem of some, but not all, "tail" implementations.)

Another technique is to open the log file every so often, collect the last *n* lines, and then close the file. This technique avoids problems when logs are rotated, but may cause the log monitor to miss important messages if the log has grown more quickly than expected (as can happen during an unusual situation). Repeated opening and closing of the file also creates substantial overhead.

The best technique, when it is possible to use it, is to instruct `syslogd` to pipe messages directly to a log monitor process. This option is not available in all implementations of `syslogd`; however, it is present in FreeBSD's syslogd and BalaBit's `syslog-ng` [11] and can be easily added to other logging daemons. Some versions of syslogd, including the Berkeley-derived Linux `syslogd` and Core-SDI's modular syslog (`msyslog`) [12], cannot pipe directly to an arbitrary program but can send output to a named pipe where an application is listening.

FreeBSD's `syslogd` makes piping to applications especially easy by handling nearly all of the logistics for the programmer. Because `syslogd` handles the logistics of distributing each log message to the required destinations (including the log monitor), the log monitor process is unaffected by log file rotation. It merely has to be prepared to save its state and exit if `syslogd` restarts (see below).

One feature of `syslogd` which may actually defeat the purpose of a log monitor is automatic output compression. When `syslogd` sees two or more identical messages bound for the same destination, it outputs the first and then counts (but does not output) the duplicates. After a predetermined delay, it outputs a message of the form "Last message repeated *n* times" indicating the number of copies received during the delay period. If still more copies of the same message arrive, the process is repeated with a longer delay between reports. Ironically, this feature -- which cannot be turned off in any version of `syslogd` known to the author -- is just the opposite of what is needed for effective log monitoring. (It is precisely when an unexpected flood of repeated messages arrives that it is most useful for a log monitor to take prompt, autonomous action. But if notice of those messages are delayed, it cannot do so.) The author has submitted a patch to the

FreeBSD Project which disables repeat counting on messages piped to a program. (With the patch in place, if the same messages also go to a file or terminal, compression will still occur on those outputs.) It may also be desirable to disable compression when logging to a remote host, since -- while this would cause an increase in network traffic -- it would facilitate the implementation of remote log monitors. Similar modifications should be made to other logging daemons that direct output either to programs or to named pipes, to facilitate the use of log monitors.

3.3 Processing Piped Output from FreeBSD's syslogd: Caveats and Tricks

While implementing his first group of trial log monitors under FreeBSD, the author learned by experience how to deal with the quirks of this particular logging daemon and operating environment. FreeBSD's `syslogd` does not start a program to which messages are "piped" until it has output for it. When it does start such a program, it executes it via `sh(1)` so that command line processing may performed. To avoid the overhead of a vestigial shell process, it is best to use the "exec" command (as shown above) to launch the log monitor script or program. The log monitor should expect to receive messages via standard input; its standard output and standard error file handles will be redirected to `/dev/null`.

Secure programming practices are of the utmost importance when one is creating log monitors. Piped applications started by FreeBSD's `syslogd` run with the same uid as `syslogd` itself -- normally root. Because a key function of log monitors is to take adminstrative action as a result of what they observe in a stream of log messages, they often must run as the superuser and may not be able to accomplish their intended functions if they "drop" privileges for safety. (In a capabilities-based system, it may be possible for a log monitor to drop capabilities that the author knows it will never use.) It is therefore especially important to avoid potential buffer overflows, format string vulnerabilities, and security holes that might arise when unfiltered input is passed directly to system APIs or programs. "Tainting" and/or extremely careful validation of input is strongly recommended.

Should an application which receives piped output from `syslogd` terminate of its own accord, it will be restarted when there is more input for it. However, syslogd may itself need to request that a log monitor terminate. (The most common situation in which this will occur is when `syslogd` receives a "hangup" signal -- SIGHUP -- indicating that it must restart and reread its configuration file.) `syslogd` indicates its desire to shut down the log monitor by closing the pipe it has created to the log monitor's standard input. The log monitor then has a predetermined amount of time -- 60 seconds in most implementations -- to save any state it wishes to preserve and terminate. If it takes longer, syslogd will attempt to kill it by sending it a "terminate" signal -- SIGTERM. To ensure that `syslogd` is able to kill a log monitor that is frozen, it is advisable for a log monitor *not* to catch SIGTERM unless it may need a very long time to save its state.

4 Log Monitors and Apache

The Apache Web server [13] is perhaps most widely deployed application for UNIX-like operating systems which does not normally log via `syslogd`. (Apache can be easily configured to log errors via `syslogd`, but there is no built-in option that allows the logging of all traffic this way.) Fortunately, Apache's own logging facilities are more powerful and flexible than those of `syslogd` itself, so logging output via `syslogd` is rarely necessary.

Apache can pipe log messages to external programs on all UNIX and UNIX-like platforms, making the implementation of log monitors, log rotators, and other such utilities very straightforward even on systems whose `syslogd` implementation does not support piped commands.

4.1 Creating Primitive Log Monitors Within Apache

Apache's logging modules also provide conditional output, pattern matching, and custom formatting. These features not only facilitate the use of

external log monitors but in some cases allow primitive log monitors to be implemented directly within Apache itself. The fragment in Listing 2, when added to the `httpd.conf` server configuration file (either in the main body or within the <VirtualHost> directives), will automatically block traffic from worms such as Nimda [14], Code Red [15], and sadmind/IIS [16].

```
# Flag requests for URIs containing common strings from Nimda-like worms
# (including Code Red, sadmind/IIS). Note that the patterns below are regexes;
# remember to escape dots and other characters with special significance!
SetEnvIf Request_URI "/winnt/system32/cmd\.exe" worm
SetEnvIf Request_URI "/scripts/root\.exe" worm
SetEnvIf Request_URI "/MSADC/root\.exe" worm
# Don't use the following patterns if you use "upreferences" in URIs
SetEnvIf Request_URI "/\.\." worm
SetEnvIf Request_URI "\.\./" worm

# Block attackers who send the patterns above within URIs. The command below
# uses a blackhole route. It's more efficient to firewall (the command
# will vary depending upon the firewall in use) or to use SSH to add rules to
# an upstream firewall to block the attacker, but this method has the
# advantage that it is relatively independent of configuration. If several
# commands must be executed, or if postprocessing of output is desired, it
# is best to invoke a script or compiled program rather than doing all the
# work from within httpd.conf.
 CustomLog "|exec sh" "route -nq add -host %a 127.0.0.1 -blackhole" env=worm

# Note that no input from the client is used in the shell command, so this
# set of directives is not subject to exploits via crafted strings. If strings
# from the client were used, stronger input validation would be in order.
```

Listing 2: Worm blocker implemented entirely within Apache's httpd.conf

While BSD and Apache cannot be "infected" by the worms targeted by the directives in Listing 2, a worm can nonetheless tax a Web server by consuming processes in the Apache process pool, glutting system logs with error messages, and infecting susceptible machines elsewhere on the network.

The configuration shown in Listing 2 uses Apache's SetEnvIf [17] directive (implemented by the module mod_setenvif) to perform regular expression matching on incoming URIs. It then uses the CustomLog [18] directive (implemented in mod_log_config) to do conditional logging based on the results of the match. When a worm is detected, Apache pipes a specially formatted "log message" -- actually a command -- to a shell for execution. (The "exec sh" may at first glance seem redundant. However, it is necessary to restart the shell -- which is initially invoked so as to accept a command as an argument -- so that it will accept commands via standard input instead.) The command creates a "blackhole" route on the host machine and locks out the attacker. If the Web server does double duty as the gateway between the Internet and an office LAN (as is often the case in small office/home office networks), blackholing the attacker will also protect the machines that sit "behind" the server. If there is a firewall upstream of the Web server, it may be desirable to replace the command that creates a blackhole route with one that causes the firewall to block the attacker.

While the author had great success with this simple log monitor (which he crafted during the early morning hours of 18 September, 2001 when Nimda began to spread), it clearly has many deficiencies. For example, it does not check to see whether an attack is coming from the Web server's own address (which can easily happen if the machine is doing double duty as a NAT router or dial-up server.) to prevent it from blackholing itself! Nonetheless, this example is a valuable proof of concept. It demonstrates that the ability to apply a general pattern matching facility to log messages, and then execute commands based on those messages, are sufficient to allow the creation of a useful, if not perfect, log monitor. More sophisticated tools -- such as languages with built-in pattern matching -- make it even easier to

write quite sophisticated agents.

4.2 Monitoring Techniques for Use With Apache

To construct effective log monitors for use with Apache, one must understand the conventions it uses when piping log output to applications. Like `syslogd`, Apache expects piped applications to be trustworthy. It runs them with the permissions accorded to Apache's master process, which usually runs as the superuser. (Apache, in its recommended configuration, maintains a single privileged "master" process which forks a pool of unprivileged processes to handle incoming requests.) As mentioned earlier, a log monitor often requires this high privilege level if it is to take autonomous administrative action. Thus, good programming practices are of paramount importance.

Unlike `syslogd`, Apache starts piped applications as soon as it has finished reading its configuration file. This gives them time to start up before receiving the first message, improving response time at the expense of overhead. If a piped application terminates, Apache restarts it the next time a message is to be delivered to it. Like `syslogd`, Apache normally uses `sh(1)` to parse command lines and set up file redirection for piped commands.

It is important to remember, when writing log monitors for Apache, that users' log formats may vary. It is therefore best to use a custom log format that the log monitor expects -- or, alternatively, to monitor the error log, whose format cannot be customized and is therefore *almost* fixed. (One aspect of the error log format can be changed via configuration: the way hosts are identified. If HostNameLookups [19] is on, domain names are output instead of numerical IP addresses.) Note that Apache allows error log messages to be sent only go to one destination. (If there is more than one ErrorLog directive, each overrides the previous ones rather than supplying an additional destination.) Fortunately, Apache also records errors in the access logs, so using ErrorLog to feed messages to a log monitor still allows a human to review messages denoting errors.

4.3 An Extensible Worm Blocker/IDS for Apache

After implementing the "quick and dirty" Apache worm blocker described above, the author decided to create a more powerful, extensible, and maintainable one. SNOBOL4 [20] [21] was chosen because of this language's prodigious pattern matching, text handling, and parsing abilities.

The 28 executable lines of SNOBOL4 in Listing 3 detect infection attempts from worms such as Code Red, Nimda , and sadmind/IIS and "blackhole" the attacking machine. Unlike the earlier example, however, this SNOBOL program completely parses, and validates the fields of, each Apache ErrorLog message before taking action. This eliminates any chance of a "false positive," which might occur if a regular expression in the earlier example happens to match part of a legitimate request.

```
* An Extensible worm blocker/IDS for Apache in SNOBOL4
* Copyright (c) 2001 by Brett Glass
* Released under the "MIT" license; see http://www.opensource.org/licenses/mit-license.html
*
* This program accepts the piped error output from the
* Apache Web server and spots lines indicating an attack
* from Nimda.A or similar worms, including Code Red,
* sadmind/IIS, and Nimda.E. It can then firewall or
* blackhole the attacking host. Add it to your Apache
* configuration by inserting a line such as
*
* ErrorLog "|exec snobol4 -b /usr/local/bin/wormblock.sno"
*
* Also, make sure that HostNameLookups is off so that the
* log messages contain numeric IP addresses.
*
* This program is designed to be easily extensible to catch
* a wide variety of potential exploits.
*
```

```
* An Apache error log message generated by the Nimda worm
* might look like this (wrapped for readability):
*
* [Thu Nov  1 12:46:07 2001] [error] [client 12.98.224.154]
*  File does not exist: /usr/local/www/data/textorics/scripts/
* ..%5c../winnt/system32/cmd.exe
*
* Build up SNOBOL patterns for Apache ErrorLog messages.
* Use the pattern "WS" for whitespace (tabs or blanks)
        WS = SPAN(' ' CHAR(9))
        DIGITS = '0123456789'
        WEEKDAY = 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun'
        MONTH = 'Jan' | 'Feb' | 'Mar' | 'Apr' | 'May' | 'Jun' |
+        'Jul' | 'Aug' | 'Sep' | 'Oct' | 'Nov' | 'Dec'
        DAYOFMONTH = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,31) *GE(NUMBER,1)
        HOUR = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,23)
        MINUTE = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,59)
        SECOND = MINUTE
        DAYTIME = HOUR ':' MINUTE ':' SECOND
        YEAR = SPAN(DIGITS)
        DATEANDTIME = '[' WEEKDAY WS MONTH WS DAYOFMONTH WS DAYTIME WS YEAR ']'
        OCTET = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,255)
        IPADDRESS = OCTET '.' OCTET '.' OCTET '.' OCTET
        ERRSTR = '[error]'
        CLIENTINFO = '[client' WS (IPADDRESS . CLIENTIP) ']'
        FILEERR = 'File does not exist:'
        FILENOTFOUNDERROR = DATEANDTIME WS ERRSTR WS CLIENTINFO WS
+        FILEERR WS REM . PATH
        DANGEROUSPATH = '/winnt/system32/cmd.exe' | '/scripts/root.exe' |
+        '/MSADC/root.exe' | "/.." | "../"
LOOP    LOGLINE = INPUT* Anchor the matching of the error message for efficiency
        &ANCHOR = 1
* We're using unevaluated expressions ("thunks") and so must do full scans
        &FULLSCAN = 1
LOGLINE FILENOTFOUNDERROR :F(LOOP)
* Scan the path, using an unanchored match, for strings betraying a worm
        &ANCHOR = 0
        &FULLSCAN = 0
        PATH DANGEROUSPATH :F(LOOP)
        HOST(1,'logger -t wormblock -pauth.notice Nimda or similar attack detected!'
+        'Blocking IP address ' CLIENTIP)
        HOST(1,'route -nq add -host ' CLIENTIP ' 127.0.0.1 -blackhole')
        :(LOOP)

* Note that a blackhole route is a brute force blocking method. It
* allows the first SYN to arrive but blocks the outgoing SYN-ACK,
* causing the TCP three-way handshake to fail. Its advantage is
* that it works on nearly any system regardless of configuration.

* If you're running a firewall, you can replace the route command
* above with ones that add firewall rules. Here are samples for
* FreeBSD's ipfw:
*       HOST(1,'/sbin/ipfw -q add deny all from any to ' CLIENTIP)
*       HOST(1,'/sbin/ipfw -q add deny all from ' CLIENTIP ' to any')
* The commands for ipf and pf are similar.
* If you want to block attacks at a different machine (say, the
* firewall that guards your entire network), you can use SSH to send
* the firewall similar commands. The exact commands required will
* depend upon your network and firewall configurations.

END
```

Listing 3: Extensible worm blocker/IDS for Apache in 28 lines of SNOBOL4

The example in Listing 3 was written for Philip Budne's free Macro SNOBOL4 for UNIX [22], which compiles on most BSD UNIX implementations and is present in the NetBSD and FreeBSD ports collections. It is readily portable to other implementations of the language including Catspaw SPITBOL [23]. The author has found SNOBOL4 to be extremely useful for log monitoring -- even more so than Perl -- because its extremely powerful recursive pattern matching allows it to completely parse a log message by executing a single line of code. SNOBOL4 also allows more extensive input validation to be done within a pattern than can easily be done within a

Perl regular expression. For example, in the patterns DAYOFMONTH, HOURS, MINUTES, and SECONDS patterns, the GE() and LE() predicates are applied to strings of digits during pattern matching to ensure that the numbers they represent are within allowable limits. SNOBOL's pattern matching engine can backtrack (or indicate failure) if these conditions are not met.

4.4 Refinements to the Initial Design

Note that the code in Listing 3, like that in Listing 2, immediately and unconditionally blocks any host which attacks the server on which it is running. As noted earlier, this could be an unfortunate administrative decision under certain circumstances. For example, if an infected dial-up user is blocked, subsequent users of that dial-up line will not be able to reach the site. If an attacking machine is behind a NAT router or a proxy, every other user arriving from the same site may be blocked. (This is a particular concern in the case of AOL, which passes all HTTP requests through caching proxies to conserve bandwidth and anonymize users.) Also, a malicious third party could post (or e-mail to users) links which, when followed, caused a block to be put in place.

Fortunately, it is relatively simple to make refinements to the log monitor shown above to handle these problems. A "do not block" list can be added to ensure that the machine does not block itself. By requiring two or more hits from an IP address before blocking it, the monitor can reduce the chances of an accidental block or of a block caused by a maliciously distributed link. Use of the MAPS Dial-up List (DUL) [24] to recognize dial-ups, plus an "amnesty" policy, can protect against long term blocking of dial-ups, though at the expense of a few more hits from worms.

Other refinements suggested during previous presentations of this work include:

- The ability to notify an administrator of the current block list (and/or "repeat offenders") so that s/he can notify administrators by phone or e-mail;
- The ability to mail or page an administrator when an attack is detected from a host on the "do not block" list (or under other conditions);
- The ability to gather statistics (such as the number of hits received from a particular Web address or subnet per hour) and automatically notice anomalies;
- The ability to place the log monitor on a separate machine, so as to preserve both copies of logs and information about attacks or malfunctions in the event of catastrophic system failure or tampering; and
- The ability to view a display and/or reports detailing the log monitor's actions.

All of these refinements are easy to add due to the flexibility of the SNOBOL4 language, which provides associative arrays, record types, indirect references, and other features typically found in high level interpreted languages.

5 Creating a Generalized Logging and Log Monitoring Facility

The goal of the author's ongoing research, however, is not to perfect any one special-purpose log monitor. It is, rather, to learn, via the creation of a diverse collection of log monitors, what features are desirable in a generalized logging and log monitoring facility. The long term goal is to create a single facility -- much more powerful than the `swatch` and `2swatch` scripts mentioned earlier -- which can subsume the functions of simple log monitors and facilitate the generation of more sophisticated ones. Such a facility should be usable across a wide range of operating system platforms, and should allow the creation of monitors via a process which consists as much as possible of configuration rather than programming. Features of this facility are expected to include:

- Compatibility with the "legacy" logging facilities and facility/severity codes of current UNIX implementations;
- The ability to apply pre-written message parsing templates to messages (akin to the "distillation" process used by Lire [25]

but performed in real time) so that rules can refer to message field by name;

- The ability to identify and report messages which were not parsed (possibly indicating an obsolete template and/or a software problem);
- The ability to access all information associated with a log message and the process that generated it -- including the identity of the program, effective user and group ids, facility and severity codes, point of origin (if not on the local system), etc.;
- Accumulation of statistics (e.g. number of e-mail messages received from a specific user or IP address) for use in rules;
- The ability to correlate log messages and statistics produced by different applications, e.g. a POP server and an SMTP server;
- The ability to generate one or more periodically refreshed displays (e.g. bar graphs) based on log statistics;
- The ability to query external databases such as DNS blacklists;
- The ability to maintain, save, and restore internal databases (e.g. of blocked hosts and times at which they were blocked) and report their contents at runtime;
- The ability to "fire" rules at specific times or intervals as well as in response to messages;
- The ability to send log messages to, and accept them on or from, arbitrary UDP or TCP ports;
- The ability to log to another machine via an encrypted connection (e.g. through SSH or SSL);
- Stronger authentication than that implemented in current versions of `syslogd` (most of which use source IP address and port number);
- Flexible notification facilities, including the ability to send notices via e-mail, pager, IRC, and instant messaging systems;
- The ability to issue commands to firewalls, routers, bridges, managed hubs, and remote power controllers; and
- The ability to allow or deny users access to facilities (e.g. by changing group memberships, changing a user's login shell to `/etc/nologin`, or removing and restoring passwords).

Input from system architects and administrators regarding suggested features and functionality is welcome.

6 Conclusions

Any computer system which is connected to the Internet, and/or subject to misuse by its users, requires constant vigilance to fend off attacks and thwart abuse. Because system administrators cannot be expected to monitor system activity 24x7, intelligent agents -- or log monitors -- can alert them to situations that require attention and "hold the fort" until help arrives. Log monitors can also perform highly routine chores -- such as blocking worms and spammers – automatically.

While a handful of "plug and play" log monitors now exist, none contain the features necessary to allow them to perform sophisticated stateful monitoring. The author's goal is to fill this gap by implementing a collection of complex log monitors and then creating a generalized facility which can subsume all of their functions.

To ease the implementation of log monitors, the logging facilities in different UNIX implementations -- which have diverged in subtle ways and often hide useful information from administrators and intelligent agents alike -- should be updated or replaced with a more modern scheme that is backward-compatible with what exists today. It will then be much easier to implement a generalized log monitoring facility that runs on a wide variety of platforms.

Of course, no log monitoring system can completely replace the insight or talents of a

human administrator. As Bruce Schneier, founder of Counterpane Network Security, writes in a white paper posted at http://www.counterpane.com/msm.html :

> *Network attacks can be much more subtle than a broken window. Much depends on context. Software can filter the tens of megabytes of audit information a medium-sized network can generate in a day, but software is too easy for an attacker to fool. Intelligent alert requires people. People to analyze what the software finds suspicious. People to delve deeper into suspicious events, determining what is really going on. People to separate false alarms from real attacks. People who understand context.*[26]

The correct approach, therefore, is not one that eliminates people but one that uses intelligent agents -- log monitors -- as a first line of defense. This frees skilled administrators from the tedium of reviewing logs, so that they may focus on the bona fide anomalies detected by log monitors and on other problems more worthy of their talents.

7 Acknowledgments

Thanks to the many attendees of BSDCon Europe 2001 who provided many useful suggestions regarding this ongoing work, and to "shepherd" Gregory Neil Shapiro who guided the preparation of this paper. Thanks also to the authors and maintainers of the BSDs and of the other utilities mentioned in this document for their contributions to the state of the art. Trademarks mentioned in this document are the property of their respective owners.

References

[1] Stephen E. Hansen and E. Todd Atkins. Centralized System Monitoring With Swatch. In *Proceedings of the Seventh Systems Administration Conference (LISA)*, Monterey, CA, Nov. 1993. Paper URL: http://www.stanford.edu/~atkins/swatch/lisa93.html . Software at URL: ftp://ftp.stanford.edu/general/security-tools/swatch.

[2] Pacific Institute for Computer Security (PICS) research group, San Diego Supercomputer Center (SDSC). 2swatch. Software at URL: ftp://ftp.sdsc.edu/pub/sdsc/security/PICS/2swatch/.

[3] Paul Vixie and contributors. cron. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=cron&apropos=0&sektion=8&manpath=FreeBSD+5.0-current&format=html.

[4] Paul Traina and Brian Somers. periodic. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=periodic&apropos=0&sektion=8&manpath=FreeBSD+5.0-current&format=html.

[5] Tom Boutell. Wusage. Software and documentation at URL: http://www.boutell.com/wusage/.

[6] Kai Schlichting. Kai's Spamshield. Software and documentation at URL: http://spamshield.conti.nu/.

[7] Eric Allman, The University of California at Berkeley, The FreeBSD Project and contributors. syslogd. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=syslogd&apropos=0&sektion=8&manpath=FreeBSD+5.0-current&format=html.

[8] Eric Allman, University of California at Berkeley and contributors. syslogd. 4.4BSD Lite2 version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=syslogd&apropos=0&sektion=8&manpath=4.4BSD+Lite2&format=html.

[9] Eric Allman, The University of California at Berkeley, The FreeBSD Project and contributors. syslogd.conf. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=syslog.conf&apropos=0&sektion=5&manpath=FreeBSD+5.0-current&format=html.

[10] Matija Grabnar. File::Tail. Software at URL: http://www.cpan.org/modules/by-module/File/File-Tail-0.98.tar.gz.

[11] BalaBit IT Ltd. syslog-ng. Software and documentation at URL: http://www.balabit.hu/en/downloads/syslog-ng/.

[12] Core-SDI. msyslog. Software and documentation at URL: http://community.corest.com/pub/msyslog/.

[13] The Apache Software Foundation. Apache HTTPD Server Project. Software and documentation at URL: http://www.apache.org/.

[14] Computer Emergency Response Team (CERT). Advisory CA-2001-26: Nimda Worm. At URL: http://www.cert.org/advisories/CA-2001-26.html.

[15] Computer Emergency Response Team (CERT). Advisory CA-2001-19: "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. At URL: http://www.cert.org/advisories/CA-2001-19.html.

[16] Computer Emergency Response Team (CERT). Advisory CA-2001-11: sadmind/IIS Worm. At URL: http://www.cert.org/advisories/CA-2001-11.html.

[17] The Apache Software Foundation. mod_setenvif. Documentation at URL: http://httpd.apache.org/docs/mod/mod_setenvif.html#setenvif.

[18] The Apache Software Foundation. mod_log_config. Documentation at URL: http://httpd.apache.org/docs/mod/mod_log_config.html#customlog.

[19] The Apache Software Foundation. Apache Core Features.

HostNameLookups directive. At URL: http://httpd.apache.org/docs/mod/core.html#hostnamelookups.

[20] R. E. Griswold, J. F. Poage, I. P. Polonsky. The SNOBOL4 Programming Language, 2nd Edition. Bell Telephone Laboratories/Prentice-Hall, 1971.

[21] Phil Budne. Phil's SNOBOL Resources Page. At URL: http://people.ne.mediaone.net/philbudne/snobol.html.

[22] Phil Budne. Macro Implementation of SNOBOL4 in C (C-MAINBOL). Software and documentation at URL: http://people.ne.mediaone.net/philbudne/src.html#snobol.

[23] Mark Emmer. Catspaw SPITBOL. Information at URL: ftp://ftp.snobol4.com/specshet.pdf

[24] Mail Abuse Prevention System (mail-abuse.org). MAPS DUL Introduction. At URL: http://www.mail-abuse.org/dul/intro.htm.

[25] LogReport Foundation. Report Production Line. At URL: http://www.logreport.org/documentation/about=architecture.

[26] Bruce Schneier. Managed Security Monitoring: Network Security for the 21st Century . At URL: http://www.counterpane.com/msm.html.

Sushi – an extensible human interface for NetBSD

by

Tim Rightnour
The NetBSD Project
garbled@netbsd.org

Abstract

This document describes Sushi, a menuing system designed for the administration and configuration of the NetBSD operating system. Included are detailed explanations of the reasons behind the creation and design of Sushi. In addition, this document includes descriptions of the file formats and examples of the menus provided by Sushi.

1. An introduction to Sushi

Sushi stands for Simple to Use System-Human Interface. Sushi was designed to provide an easy to use environment, allowing a new administrator to set up or maintain a NetBSD system. Sushi can be used for a variety of tasks, such as the initial configuration of a system or to facilitate maintenance. Custom menus can be created and installed at any time, allowing administrators to create their own menus to perform frequently used tasks.

Sushi is a text-based user interface to the system. It has been written in C and carries a BSD license. Tasks that have already been written for Sushi include configuring network interfaces, configuring startup (rc) scripts, user management, and installation of third party packages using the NetBSD package system.

Currently, Sushi is only available in the development branch of NetBSD and has not yet been made part of an official release. Administrators wishing to try out Sushi can do so by installing one of the NetBSD-current snapshots or by compiling it from source. Installation on an older system is unlikely to work, as the provided menus are tuned for the development branch and Sushi itself requires many curses features and related libraries not available in the 1.5 release of NetBSD, such as *libform* and *CDK*.

When starting Sushi the administrator is presented with a menu of task categories. The administrator can select one of these categories and will then be provided with additional options, either more categories or actual tasks will be presented. Once a task has been selected the administrator will then be presented with a form.

A form usually comprises a series of questions, with blanks next to each one for providing answers. A typical form might have configuration details, such as the IPv4 address of a network interface to be setup or possibly a series of yes/no questions to provide options. Once the administrator has edited this form it can then be submitted to be processed by Sushi.

Processing the form generally consists of interpreting the various options and fields the administrator has filled out and executing the proper actions. For example, had the administrator configured a network interface, Sushi would apply those changes to the interface. As the form is processed, a window with the output of the commands Sushi is executing will be displayed. If the commands fail, the administrator is informed and can then edit the form again. Sushi makes no attempt to interpret the output of the command that has been executed, the analyzation of the failure is left to the administrator. Once the task has been completed, the administrator can go on to other menus and perform additional tasks or exit Sushi.

2. Design Goals

Sushi was designed with a number of specific goals in mind. First, it needed to provide an intuitive interface to the system. Second, it needed to be easily extensible so virtually any task could be programmed into Sushi, without recompiling or having to learn another programming language. Third, it needed to provide support for a variety of native languages. Finally, it needed to provide for compatibility with manual changes made to the system.

2.1. Simple to Use

In order to be useful to most new administrators of the NetBSD operating system, Sushi needed to have a very instinctive interface. If the tool is just as or more complex than the actual system is to use, then there is no point in using it. The instinctive interface was accomplished by setting up a simple hierarchical menu structure. This allows tasks to be easily categorized by their general function or area of influence. For example, under a menu called "User and Group Management", one might place functions such as creating and deleting users, assigning users to groups, and modifying users or groups.

The interface also allows a number of input field types that define the types of data the administrator can enter. One such field is the basic text entry, where freeform text can be entered. Information such as host names or data files names can be entered by the administrator here. Other field types include multiple choice selection fields, where the administrator can pick from a predetermined set of choices or restricted fields, where the administrator can only enter things such as numbers or IPv4 addresses.

Sushi also attempts to provide help for all menus or tasks. A simple status bar at the bottom of the screen shows some of the basic navigation commands available to the administrator. In most menus or tasks, the help key (F1) will bring up a section of help text written specifically for the current menu or task. This help text is designed to explain to the administrator what some of the various questions or options mean and assist in choosing the appropriate selection. It often points to manual pages, which can assist the administrator further in determining how to proceed.

Sushi attempts to provide this easy to use interface without becoming overly cumbersome. The features which Sushi can provide are useful to administrators of all skill levels, not because the tasks can't be accomplished by hand, but because the menu interface is easier than editing the files by hand. Sushi attempts to provide the menu and form to the administrator, without forcing them to answer too many questions or fill in unnecessary fields.

2.2. Easily Extensible

While a menuing system written entirely in a compiled language might provide an easy to use interface for the administrator, it does so at the price of maintenance headaches. Sushi needed to be easily modifiable and provide a way for administrators to quickly add menus of their own to the system.

One of the reasons Sushi is able to provide an easily extensible menu interface is because it is an interpreter, rather than a precompiled set of menus. All of the menus and tasks that are provided by Sushi are actually taken from a set of command files and directories living on the machine.

At start up, Sushi looks in a number of base directories for its index files. An index file contains a description of a sub-menu or task and the name of a subdirectory where that sub-menu or task lives. It also contains a "quicklink" keyword which can provide a fast method to jump to the menu from the command line. Subdirectories can contain more index files or they can terminate, in what is called an "endpoint".

An endpoint is a task that is to be completed. It can be anything from a command that is executed when the administrator selects the menu item, to a complex set of forms that need to be filled out by the administrator. They can even point to functions internal to Sushi, allowing very complex tasks to be performed or the modification of internal Sushi variables, such as turning on logging.

Forms are specified by a form file. This file describes the form to Sushi, indicating field types, field descriptions, and data specific to the field, such as predetermined choices for a multiple choice field.

Besides administrator input, fields can also gather data from multiple sources. For example, if we were writing a form to modify the machine's network interface, we would want to provide the administrator with the current settings for that interface when the form is displayed. This can be accomplished by using scripts. Scripts are basically any executable program that outputs the desired information. Arguments can be passed to these scripts, allowing the menu to tell the script which network interface is being configured. By passing these arguments, Sushi can call the script that

can look up the IPv4 address of the interface and read the data back to fill in the field. When the form is displayed to the administrator the current IPv4 address will be present in the field, either to edit or leave as is.

In the case of a form, a script which is executed upon completion of the form also exists. The script can be any type of executable program, giving freedom to the designer to use the language most suitable for the task. This script is executed and given the contents of each field, in order, as its arguments. The script is expected to verify the choices made by the administrator and execute the appropriate task. For example, with the form that configures a network interface, the script would analyze the arguments given from the form and construct an ifconfig command and execute it. The exit code of the script determines the success or failure of the action to be displayed to the administrator watching the output screen.

The base directories that Sushi searches for index files can also be modified through a configuration file by simply adding more directories to the list. In addition, one of the directories searched by Sushi is the administrator's home directory. This allows administrators to create menus that are only available to the creator and store them locally. For example, an administrator might wish to create a menu to automate a personal task, such as customization of a personal rc file. Using this mechanism, third party packages can be extended to install Sushi menus for their own configuration. When Sushi scans for its index files, it creates a single hierarchical tree out of all the menus it finds.

2.3. Internationalization Support

Sushi also provides support for multiple written languages. All of the text displayed by the Sushi engine itself was written using the *catgets(3)* interface and is therefore capable of supporting a limited set of native languages.

In addition, most of the data files used by Sushi (such as the help text, forms, and index files) can all be written in an alternate language and stored using different file suffixes. When Sushi is started in an environment where the preferred user interface language variables have been set, it will load the proper files whenever they are available and failing that display the English defaults.

2.4. Compatibility with manual changes

When making changes to system configuration files, it is important to do so in a way that is compatible with manual changes. Many automated configuration systems control their files by placing special markers in the file, causing an administrator to have to work around these markers. Other programs might require that the files only be modified by the configuration program and never edited by hand. This is unacceptable for a configuration tool like Sushi for a number of reasons.

First and foremost, Sushi is designed to assist a new NetBSD administrator in getting his or her system up and running quickly. Once the new administrator has overcome the learning curve of using NetBSD and is competent in it, they should not be penalized for having used Sushi or be forced to continue to use it.

Secondly, a machine might be administered by multiple people or change hands. Sushi should not leave the system in an unmaintainable state, nor should it be unable to cope with changes made manually, if the new administrator wishes to use Sushi to edit previously handmade configuration files.

Sushi retains compatibility with manual changes by having a basic understanding of the various configuration file formats. Some file formats are more complex than others and in some cases certain options that are available are impossible to implement in a script. However, Sushi makes all possible efforts to interpret files properly and write them back in a readable format. Sushi also does not use any special markers or create uneditable entries in the files. This is not a feature of the engine, but rather a feature of the menus that ship with Sushi as part of NetBSD. It is also considered a golden rule when creating new menus for Sushi.

3. Designing menus for Sushi

The programming interface for Sushi has been designed to be very simple to pick up and begin writing menus with. Generally speaking, if an administrator can write a script to accomplish a specific task, they can turn that work into a Sushi menu with a minimal amount of work.

3.1. Search order

Upon entering a directory, Sushi looks for a number of different files specifying the action it should take. Sushi looks for these files in a specific order and will begin processing the first file that it recognizes, ignoring the rest of the files in that directory. It causes an error when an endpoint has no files in it and Sushi will exit if it encounters one. The search order of these files is as follows:

- The index file "index".
- The preform file "preform".
- The form file "form".
- The script file "script".
- The execute file "exec".
- The function file "func".
- The help file "help".

For each file, Sushi will first attempt to load a file ending in a locale suffix, such as ".de" for German. Should it fail to find the appropriate translated item, it will then search for the file without a suffix.

3.2. The index file

The index file comprises multiple lines containing three whitespace separated columns where each line of this file represents a single menu item. These are the name of the subdirectory containing the menu item, a quicklink, and finally a description of the menu item.

The subdirectory argument is used to specify which subdirectory contains the next sub-menu or endpoint. Any subdirectories specified in the index file must exist for Sushi to process the tree properly. The subdirectory can be replaced with the keyword "BLANK" to place a blank line on the menu. This can be used to group certain types of actions together in the menu.

The second argument to a line in the index file is the quicklink. This is a single word that can be used to jump to this menu or task from the command line. It should be something that is easy to remember and obvious to the administrator, such as "users" to point to the "User and Group Management" sub-menu. The administrator can then jump directly into this menu by starting Sushi with "users" as the only argument. Again, the "BLANK" keyword can be used to specify a blank line in the menu.

The description of the menu item is meant to give a brief title to the sub-menu or task located in the subdirectory below. It should consist of a brief title, such as "User and Group Management". Should the entry point to a sub-menu, the description text will be used as the main heading for that sub-menu. This title is limited to approximately 70 characters to allow it to fit on a standard 80 column wide screen. Again, the description can be replaced with the "BLANK" keyword to create a blank line.

It is important to note that when creating blank lines in the menu that all three arguments must contain the "BLANK" keyword. Sushi also ignores any lines beginning with a comment symbol (#). For an example of an index file see figure 1.

3.3. The form file

A form file is a whitespace delimited list of fields consisting of a field type and followed by a description. Fields may be of many different types and each one has a set of different arguments it expects. Arguments are separated from the field type keyword with a ":" and separated by commas.

There is also a "preform" file, which can be used to provide a series of forms to the user. The preform allows the programmer to gather data from the user, which can later be used in the form to populate certain fields. An example of where this might be useful would be setting up a network interface. The preform would ask the administrator which interface they wanted to setup and pass that information to the form so it could query the proper interface for its current settings. Any data entered into a preform is made available to the form via the special argument key "@@@1@@@", which specifies the data from the first field of the preform. When Sushi interprets the form file, the special argument key will be replaced with the data the administrator entered whenever it is found in the form file. Sushi is limited to a single preform and associated form pair. Multiple preforms are not possible.

The first type of field is the basic free-form entry field, which is denoted by the keyword "entry". The only argument for this keyword is the length of the maximum entry in characters. When the entry field is specified, a

```
# $NetBSD: index,v 1.5 2001/04/26 02:26:16 garbled Exp $
install         install Software Installation and Maintenance
system          system  System Maintenance
users           users   Security and Users
procs           procs   Processes and Daemons
network         network Network related configuration
BLANK           BLANK   BLANK
info            info    Using sushi (information only)
util            util    Sushi utilities (logging/scripting)
```

Figure 1. Example of an index file

blank underlined input field will be placed on the display into which any type of data may be entered. You may also prefix the entry keyword with "req–" to specify that the field must be filled in by the administrator before form processing can take place. Required fields are prefixed with an asterisk on the display.

An "escript" field type is an entry field whose initial value is filled in by running an associated script. The arguments to an escript field type are the maximum field length, the name of the script to be executed, and any optional arguments you wish to pass to the script. The script that is executed can be of any executable format and is expected to return a single line of text. It is important that the script always return something to the Sushi engine to avoid possible errors at run time. The escript keyword can be prefixed with "req–" to make it a required field or be specified as "nescript" to create an uneditable field. Uneditable fields may be used to display data to the administrator without allowing them to modify it. Data in these fields will still be passed to the task script upon completion of the form.

A list field type is specified by the keyword "list". This field type will present the administrator with a multiple choice field. An administrator can only select one of the predetermined choices from the list and may not modify any of them. The administrator can toggle the values of the list by using the TAB key or bring up a selection list box containing all of them with the F4 key. The arguments consist of a comma separated list of possible choices. This is especially useful for generating yes or no questions for the administrator. The list keyword can be prefixed with "req–" to make it a required field.

A multilist, specified by the "multilist" keyword, is a list where the administrator can select more than one of the possible choices from the list. This is accessed by the administrator via the F4 key and selections are toggled with the space bar. The format for the multilist is the same as a list.

The "blank" keyword can be used to create an item with no corresponding entry field. This can be used to provide additional lines of description for the previous field. No data is passed to the task script when a blank field is specified and a blank field does not count as an argument to a task script.

The "noedit" keyword can be used to create an uneditable field that will still be passed to the task script. This is similar in operation to the "nescript" field type and is used primarily in displaying data to the administrator, or passing special arguments to the task script. The only argument is the string to be displayed in the field.

The "invis" field type allows the programmer to create fields which will not show up on the form. The description for the field will still be visible however. The contents of the field will be passed to the task script upon

completion of the form. The only argument is the string to be placed in the invisible field.

A function field, specified by the "func" keyword, allows the programmer to create special list field types whose values are populated by calling a function internal to the Sushi engine. In order to utilize the function field type, the appropriate function must first be programmed into the Sushi engine and the engine must be recompiled. The function field has two arguments, the name of the function to be called and a single text argument to be passed to the function. Functions are expected to return a list of values and have the following prototype:

*(char **)function(char *argument);*

Functions must be added to the "functions.c" source file and added to the "func_map" array at the top of that file, as well as to the "functions.h" header file. The function keyword may be prefixed with "req–" to make it a required field. Multilist functions are possible by specifying the keyword "multifunc" and are programmed identically to function fields. It is strongly encouraged that functions be avoided whenever possible in Sushi. Functions require recompiling of the engine in order to be made available and are therefore more difficult to maintain. Whenever possible other field types should be used.

The "script" field type, allows the programmer to create a list-type field, whose contents are created by running an executable program. The arguments to the script field type are the name of the script to be executed and any number of arguments the programmer wishes to pass to that script. The script is expected to produce a list of values, one per line, on standard output. These values are then read and used to create the list field type. The script keyword may be prefixed with "req–" to make it a required field. Multilist scripts can be made by specifying the keyword "multiscript". The named script is expected to be located in the same directory that the form file exists in.

To restrict field input to an integer type the "integer" keyword can be used. This field type has three required arguments and an optional fourth argument. The first three arguments are, in order, the maximum length of the field in characters, the minimum integer value allowed for this field, and the maximum integer value allowed for this field. The fourth optional argument is an integer default value for this field. Administrator entries in fields of these types will be checked by the Sushi engine to make sure they are between the minimum and maximum values for the field type. The integer keyword may be prefixed with "req–" to make it a required field.

Integer fields may be prefilled with data by using the "iscript" field type. This field type is similar to the escript field type, in that it executes a script which provides a value back to Sushi to fill in the field. The arguments for the iscript field type are the maximum length of the field in characters, the minimum integer value allowed for this field, the maximum integer value allowed for this field and the name of the script to execute, as well as any number of additional arguments the programmer wishes to pass to the script. The iscript field type can be made into a required field by prefixing it with "req–".

Fields may also be restricted to IPv4 and IPv6 addresses through the field types "ipv4" and "ipv6". Each of these may be prefixed with "req–" to make it a required field. The IPv4 field type allows addresses to be entered in dotted quad format or in hex. Both of these field types have one argument, which is the optional prefilled value for the field.

In addition, there are script forms of the IPv4 and IPv6 field types, called "ipv4script" and "ipv6script." Both of these may be made into required fields by prefixing them with "req–". The arguments to these fields are the name of the script which is to be executed and any optional arguments the programmer wishes to pass that script. These fields both behave similarly to the escript field type.

When programming a field that has optional arguments, such as the ipv4 field type, it is important to still use the ":" symbol to separate the field keyword from the arguments. In order to produce an ipv4 field which has no data prefilled into the field the programmer would create a line such as:

```
ipv4:    IPV4 address
```

By using combinations of the above field types, it is possible to create nearly any type of form that the programmer may wish to build. Going back to the previous example of creating a Sushi menu to configure network interfaces,

figures 2 and 3 are the preform and form files to accomplish this, respectively.

In the preform file (figure 2), a script is being run which collects the names of different interfaces on the machine. This script could be something like:

ifconfig -l | xargs -n 1 echo

Once the administrator has selected an interface from the list, that interface name will then be passed to the form.

In the form file (figure 3) each instance of "@@@1@@@" will be replaced with the network interface the administrator had selected from the preform, say for example "fxp0". The fourth line of the form asks the administrator for the IPv4 address of the interface. The ipv4script field type will run the script "script2" giving it the arguments "4" and "fxp0". In this case, the script2 script, runs "ifconfig fxp0" and pulls the address out of that, prefilling the field for the administrator with the IPv4 address of the fxp0 interface.

3.4. The script file

The script file is a script or executable program of some type, that is executued by Sushi when encountered. The script can be encountered in one of two ways.

First, if there are no other files found in the search order, the script will be run when the

```
# $NetBSD: preform,v 1.1 2001/04/25 03:43:33 garbled Exp $
script:script1                Select an interface to operate on:
```

Figure 2. Example of a preform file

```
# $NetBSD: form,v 1.1 2001/04/25 03:43:33 garbled Exp $
noedit:@@@1@@@                Changing interface:
list:both,now,boot            Modify interface at boot-time, now, or both?
req-ipv4script:script2,4,@@@1@@@      Interface IPV4 Address
ipv4script:script2,n,@@@1@@@          Interface IPV4 Netmask
ipv4script:script2,b,@@@1@@@          Interface IPV4 Broadcast Address
script:script2,m,@@@1@@@              Media Type
script:script2,o,@@@1@@@              Media Options
ipv6script:script2,6,@@@1@@@          Interface IPV6 Address
iscript:3,0,128,script2,pre,@@@1@@@   Interface IPV6 Prefix Length(netmask)
escript:32,script2,i,@@@1@@@          Interface Network-ID
multilist:link0,link1,link2           Interface link options
iscript:5,1,99999,script2,mtu,@@@1@@@ Interface MTU
iscript:2,0,99,script2,met,@@@1@@@    Interface Metric
```

Figure 3. Example of a form file

menu item is selected. No arguments will be passed to the script when run in this manner. Output of the script will be displayed to the administrator and success or failure status will be noted.

The second way a script file can be executed is in response to completing a form. When a form has been filled out and accepted by the administrator, Sushi will execute the script file giving it the data filled in the form as arguments. Empty fields will be passed to the script as empty string arguments, ensuring that field position will always remain the same when translated to arguments.

The second form of the script file is generally where the actual actions take place in Sushi. In the example of modifying a network interface, the script would interpret the data from the form and make the actual changes to the network device or startup files. The script should make all attempts to fail cleanly with an error code of 1. Success should be noted with an exit code of 0.

3.5. The execute file

The execute file "exec" can be used to execute a simple program located anywhere on the system. It is generally used to provide simple access to programs that provide information about the system. The program will be executed as it appears in the execute file, with no arguments being passed or interpreted.

An example of using the execute file, would be to program a simple menu which displayed a list of packages installed on the administrator's system, in which case the execute file would contain: "pkg_info".

3.6. The function file

The function file "func" can be used to provide access to some of Sushi's internal functions that are made available to the menu programmer. The format of this file is the name of a function, followed by a comma and an optional single argument which will be passed to the function as a string.

This can be used to activate simple features in Sushi or program more complex ones. One example of using this, is to turn on the internal logging functionality of Sushi, which writes all actions out to a logfile. This is accomplished by having a function file containing: "log_do,on".

3.7. The help file

The help file is a simple text document that will be displayed to the administrator when the help key is pressed (usually F1). The help file can be used to give the administrator more information about what the various menus available do or what the individual fields mean in a form. A help file can be located anywhere in the Sushi tree.

The help file should give a brief description of the menu items or form it is describing. It should not be used to completely explain a subject matter to the administrator, rather it should point him to documents or manual pages which provide that information. It is a good idea to provide help files for every menu in the tree, to allow administrators to understand what each item does and warn them of potential pitfalls that may lie ahead.

4. The future of Sushi

Sushi still has a number of goals left to accomplish before it can be considered complete. Additional menus and tasks still need to be written. Eventually, most administration tasks that need to be performed on a NetBSD system will be automated in some way by Sushi. Certain areas of Sushi still need enhancement, such as displaying the command that will be executed to the administrator before executing it (to aid a new administrator in learning NetBSD commands).

In addition, due to the design of Sushi, it would not be difficult to write other processing engines with a different interface. For example, a web-based interface or X11 interface could easily be written for Sushi, reusing most of the parsing code.

5. Conclusion

Sushi was written to provide a intuitive and easy to use interface to the NetBSD operating system. It is my hope that as Sushi evolves and encompasses more tasks that administrators do on a daily basis or while setting up a machine for the first time, more users will find the learning curve of NetBSD less daunting. Sushi will one day allow a novice administrator to completely administrate a machine, making NetBSD a more user-friendly

operating system in the process.

6. About the Author

Tim Rightnour has been a user of NetBSD since 1994 and has been a developer for NetBSD for approximately 3 years. He has worked on projects ranging from device drivers to the NetBSD Package System. He is also the author of ClusterIt, a collection of programs used to automate and administer large groups of machines.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association

Addison-Wesley
Kit Cosper
Earthlink Network
Edgix
Interhack Corporation
Interliant
Lessing & Partner
Linux Security, Inc.
Lucent Technologies
Microsoft Research
Motorola Australia Software Centre
New Riders Publishing
Nimrod AS
O'Reilly & Associates Inc.
Raytheon Company
Sams Publishing
The SANS Institute
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
Taos: The Sys Admin Company
TechTarget.com
UUNET Technologies, Inc.

Supporting Members of SAGE

Certainty Solutions
Collective Technologies
Electric Lightwave, Inc.
ESM Services, Inc.
Lessing & Partner
Linux Security, Inc.
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Publishing
O'Reilly & Associates Inc.
Raytheon Company
Remedy Corporation
RIPE NCC
Sams Publishing
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-880446-02-2